1994


Youngsuck Cho
Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_disstheses

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_disstheses/5787

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A multi-faceted software reusability model: The Quintet Web

Cho, Youngsuck, Ph.D.
The Louisiana State University and Agricultural and Mechanical Col., 1994
A MULTI-FACETED SOFTWARE REUSABILITY MODEL: THE QUINTET WEB

A Dissertation

Submitted to the Graduate Faculty of the Louisiana State University and Agricultural and Mechanical College in partial fulfillment of the requirements for the degree of Doctor of Philosophy in The Department of Computer Science

by

Youngsuck Cho
B.A., Sogang University, 1978
MLIS, Louisiana State University, 1988
August 1994
To my parents,
Mr. and Mrs. Sung Hyun and Hung Sang Yoo Cho,
and
parents-in-law,
Mr. and Mrs. Soo Hong and Gyoo Ok Whang Min,
for their
love,
devotion,
and
encouragement
ACKNOWLEDGEMENTS

I would like to extend my sincere appreciation to Dr. Doris L. Carver for her patience, encouragement, and guidance in all stages of my study. I will always be thankful for her kindness and support. I also express much gratitude to all members of my academic advisory committee: Dr. Bert R. Boyce, Dr. Peter P. Chen, Dr. Sitharama S. Iyengar, Dr. Donald H. Kraft, and the Graduate Dean's Representative Dr. Yalcin Acar for their valuable advice and service.

I, especially, would like to extend my deep gratitude and love to my wife, Tongjoo Min Cho, for her love and our being together. Without her love and support I should have encountered many more frustrations during my study and my life. My beloved sons, Jaekun Andrea and Yoonkun Oliver Cho, have always been angels who removed the shadows in me. I am proud of them for their endurance during the ten years of my study.

I owe many thanks to my brother-in-law and my sister, Ja Whan and Young Hee Cho Ghoo, for their concern and care for our parents during my study far from home. I thank my youngest sister Moon Kyung Cho, for her blessings and for being with our parents during my absence.
I also want to extend my sincere appreciation to my brothers- and sisters-in-law, Ki Chun and Tong Hyun Min Nam, Tong Ik and Ae Kyung Lee Min, Sung Kun and Tong Sook Min Kim, and Tong Hae Min, for their support and concern. Their visits and phone calls refreshed and renewed my spirit.

I am deeply grateful to Mr. and Mrs. Hyung Sool and In Shil Kim, for their friendship and support. Without their help in purchasing a computer my research would have been slower.

I would also like to thank the Dean and all the staff of the Graduate School at Louisiana State University for their friendship, appreciation of my work as their computer analyst, and five years of financial support.

Finally, and repeatedly, I thank my parents and parents-in-law from deep in my heart for their devoted support and endless love.
# TABLE OF CONTENTS

ACKNOWLEDGEMENT ........................................... iii

TABLE OF FIGURES ........................................ vii

ABSTRACT .................................................. ix

1. INTRODUCTION .......................................... 1
   1.1 Problem Statement ................................ 1
   1.2 Objectives ........................................ 11

2. LITERATURE REVIEW .................................... 12
   2.1 Definition of Reusability ......................... 12
   2.2 Brief History of Reusability .................... 15
   2.3 Reasons for Reusability .......................... 19
   2.4 Object of Reuse .................................. 22
   2.5 Software Reuse Process ......................... 24
   2.6 Trends of Software Reusability ................. 26
   2.7 Types of Software Reusability .................. 28
   2.8 Summary ........................................... 38

3. A MODEL FOR REUSABLE SOFTWARE .................... 40
   3.1 Definition of Global Terminologies ............. 41
   3.2 Definition of Global Notations ................. 45
   3.3 Observations ...................................... 47
   3.4 Quintet Web: A Model for Reusable Software .... 53
      3.4.1 Semantic Web ................................ 55
      3.4.2 Horizontal Web ................................ 64
      3.4.3 Vertical Web ................................ 74
      3.4.4 Syntactic Web ................................ 80
      3.4.5 Alternative Web .............................. 86
      3.4.6 Summary ...................................... 90
      3.4.7 Creation of Reuse Support Information .... 92

4. IMPLEMENTATION OF THE QUINTET WEB .............. 96
   4.1 Environment and Characteristics ................ 96
   4.2 Reuse Support Information in the Quintet Web ... 97
      4.2.1 Information File Structures ................. 99
      4.2.2 Reusable Software Database Files .......... 103
   4.3 Design of the Quintet Web ....................... 106
   4.4 Design of the Semantic Web and the Horizontal Web 109
   4.5 Design of the Vertical Web ..................... 110
   4.6 Design of the Alternative Web .................. 113
   4.7 Design of the Syntactic Web .................... 114
   4.8 Global and External Functions ................. 115
   4.9 Prototype of the Quintet Web ................. 115

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
5. SUMMARY AND CONCLUSIONS ................................................. 123
  5.1 Summary of the Features of the Quintet Web .................. 124
  5.2 Contributions of the Research ................................. 133
  5.3 Future Research .................................................. 135

REFERENCES ........................................................................ 137

VITA .................................................................................. 170
# TABLE OF FIGURES

1.1 Software Development without Software Reusability 4

1.2 Implicit Software Reusability between Similar Projects .......................... 5

1.3 With Explicit Software Reusability ..................................................... 6

1.4 Software Reusability in the Time Span .............................................. 8

1.5 Software Reuse Lifecycle ................................................................. 9

3.1 Definitions .......................................................................................... 47

3.2 A Quintet Web: an Abstract View ...................................................... 53

3.3 Layers in the Semantic Web ............................................................... 60

3.4 Semantic Web ....................................................................................... 62

3.5 Links from Keywords to User Specification and Requirement Specification Phases ............... 62

3.6 Links from Keywords to Pseudo-code and Code Phases 63

3.7 Links from Keywords to Design Phase ............................................... 64

3.8 Horizontal Web ..................................................................................... 65

3.9 Multiple Layers in Horizontal Web ..................................................... 67

3.10 Direct and Indirect Links in the Horizontal Web 71

3.11 Vertical Web ......................................................................................... 77

3.12 Inheritance of Attributes in the Vertical Web ...................................... 79

3.13 Syntactic Web ....................................................................................... 82

3.14 Alternative Web ................................................................................... 88

3.15 Reuse Support Information ................................................................. 95

4.1 Information for the Quintet Web .......................................................... 98

4.2 Classes in the Quintet Web ................................................................. 108

4.3 Initial Screen of the Quintet Web Prototype ....................................... 116

vii

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
4.4 Keyword List ........................................ 116
4.5 Attribute Tag ........................................ 117
4.6 Choice of Phases .................................. 118
4.7 Choice of Webs .................................... 118
4.8 Options for Selecting a Variable Name ........ 120
4.9 Variable List ...................................... 121
4.10 Chain of a Variable $fp$ ......................... 121
ABSTRACT

Over the past decade, problems in software development and maintenance have increased rapidly. The size of software has grown explosively, complexity of software applications has increased, the nature of software applications has become more critical, and large quantities of software to be maintained have accumulated. During the same time, the productivity of individual software engineers has not improved proportionally.

Software reusability is widely believed to be a key to help overcome the ongoing software crisis by improving software productivity and quality. However, the promise of software reusability has not yet been fulfilled.

We present a multi-faceted software reusability model, a Quintet Web, that enables designers to reuse software artifacts from all phases. The Quintet Web consists of links of five types of reuse support information among existing document blocks: semantic, horizontal, hierarchical, syntactic, and alternative relationships. The five types of reuse support information are named the Semantic Web, the Horizontal Web, the Vertical Web, the Syntactic Web, and the Alternative Web. The Semantic Web defines the operational functionality of each software block. The Horizontal Web links functionally identical blocks of all phases. The Vertical Web identifies
hierarchical relationships of software blocks. The Syntactic Web forms a chain from the declaration of each variable to its uses for all variables. The Alternative Web enables software developers to select an alternative algorithm for an identical functionality.

The Quintet Web supports both software development and maintenance. When the Quintet Web is applied to development process, it provides software developers a means to check the consistency of the software being developed. The Quintet Web model is independent of a specific software development method.
1. INTRODUCTION

From the late 1960's, software engineering has gradually become a recognized discipline. However, many promises of software engineering have not yet been fulfilled. The majority of software is still being developed, instead of being engineered, and the majority of software being developed does not involve the reuse of existing software. Thus, software productivity has not rapidly increased.

Software reusability is believed to be a promising solution. The most traditional scheme, i.e., reuse of code, has not been as significant as originally believed.

The goal of this research is to design a software reusability model that can be applied to both software development and software reusability. The model should be defined in such a way that automated support is feasible. To achieve the goal, we survey the definition of software reusability in order to establish a working definition, history of software reusability, as well as the reasons, methods, and subjects of software reuse.

1.1 Problem Statement

Over the past decade, problems in software development and maintenance have increased rapidly. The size of software has grown explosively, complexity of software
applications has expanded, and the nature of software applications has become more critical. [WONG 1988] Large software products have many different possible states that they are very confusing for the developer to understand, remember and manage. [JONEGW 1990] However, an operational software industry must maintain the ability to develop complex software systems. G. Fisher claims that there is ample evidence from other industries that complex artifacts can not be created from scratch. [FISH 1991] Naturally, demand for qualified software professionals has increased. [HORO 1989] In the U.S., for example, demand for qualified personnel increases 12% each year, but the supply increases by only 4%. Moreover, the productivity of individual software engineers increases by only 35% each decade. In addition, there are large quantities of existing software to be maintained. As the result, software costs have highly increased although the quality of software has not improved proportionally. In 1986, the U.S. government spent $36 billion for software development and maintenance. [JONEGW 1990] On the other hand, software development tools and methodologies have not continued to dramatically improve the ability to develop software. [HORO 1989] This lack of integrated software development environments has hampered the development of high quality software. [WONG 1988] Therefore, software maintenance costs have also drastically
increased. According to Horowitz and Munson, maintenance takes approximately three times the cost of the original system. [HORO 1989] Suffering all these difficulties, software productivity has increased only 3 - 8 percent a year over the last 30 years [TRAC 1988a] while the price/performance ratio of computing hardware has been decreasing about 20 percent per year and the total installed processing capacity is increasing at better than 40 percent per year. [NEIG 1989] This difficulty, i.e., the problem of building large, reliable software systems in a controlled and cost-effective way, is referred as a software crisis. [KRUE 1992]

Software reusability is widely believed to be a key to help overcome the ongoing software crisis by improving software productivity and quality. [BIGG 1987] In other words, if software reuse principles are successfully applied, software development and maintenance cost should go down. G. Jones emphasizes that software reuse is the most economical method of creating new software. [JONEGW 1990] Also, better quality software and easy maintenance can be expected. Horowitz and Munson believe, "[reusable software] has the potential for increasing software productivity by an order of magnitude or more." [HORO 1989]

However, it is unrealistic to expect that every software system can be developed solely using existing
software. New computer technology and the demands of an advancing society require new and more complicated software. [HORO 1989] The demand for software increases by 900% each decade. [JONEGW 1990] Software must be continuously developed. However, software development without reusability can be reduced if we effectively reuse existing software.

![Diagram](image)

Figure 1.1 Software Development without Software Reusability

Figure 1.1 represents software development without software reusability. There is no explicit relationship between existing software and the current software development project. We cannot expect any type of explicit
software reuse. Productivity or quality improvement cannot be gained from previous experience.

![Diagram: Implicit Software Reusability between Similar Projects](image)

**Figure 1.2** Implicit Software Reusability between Similar Projects

Figure 1.2 is the situation in which one software development project has similarity with another project, but the reuse activity is still implicit. This case is a special case of the situation in Figure 1.1. We may expect savings from past experience, but the expectation is very vulnerable because there is no rigorous method or tools to
reuse. Savings depend solely on the experience and intuition of software developers.

Figure 1.3 depicts how the reuse of existing software benefits the development of new software. The intersection of existing software and current software development projects results in savings. The amount of savings depends in part on how much reuse is accomplished. The cost of the effort devoted for reuse activity must be deducted from the savings. So, in order to maximize the savings, not only the
amount of existing software artifacts reused must be maximized, but also the cost of effort devoted to the reuse activity must be minimized. The quality of existing software artifacts affects the quality of software under development. A means to control the quality of existing software as well as the software under development must be provided.

In broad terms, software reuse implies a simple activity of reusing existing software. However, software reuse is different from software salvaging or carrying-over code. Rather, software reuse must be "the process of reusing software that was designed to be reused." [TRAC 1990]

Figure 1.4 illustrates the software reuse process over time. A current software project requests reusable software from the existing software repository and receives software fragments. When the current software development project ends, the outcomes of the project will be merged to the software repository and be a part of existing software. Future reuse requests will be made and reusable software will be expected in response for the software development projects in the future.
This can be illustrated as the life cycle of software in terms of software reusability in Figure 1.5. A great majority of currently existing software was not designed to be reused. So, we used different terms: existing software artifacts and reusable software. Existing software artifacts simply means any existing software, while reusable software is the software designed to be reused. If we develop not just software, but reusable software, the proportion of reusable software to existing software artifacts will grow larger.
For a reuse system to be useful, the system must provide many building blocks, but when many building blocks are available, locating and selecting an appropriate one becomes a difficult problem. [FISH 1991] Retrieval and selection problems are technical problems of a different dimension. They may be exploited separately. Since reusable software must be designed to be reused, the retrieval problem must be considered when it is developed. Our point here is that a software repository must provide enough software artifacts so that when a software developer...
needs software artifacts to reuse, they can be found in the repository. In spite of the retrieval and selection problems, the more reusable software that is in the software repository, the greater the chance that software will be reused. In addition, the retrieval and selection problem is being solved gradually in many organizations. Numerous examples can be found that provide evidence that software objects can be retrieved and adapted.

One example of this retrieval and adaptation is a high-functionality computing system. [FISH 1991] As another example, a statistical report about a four-year experiment in NTT assures that these problems can be solved. According to the report, the average reuse ratio increased every year. The reuse ratio of 3% in the first year jumps to 12% in the second year, and then gradually increased to 17% in the third year and 20% in the fourth year. The average size of reused modules also went up each year. The software development projects increasingly used more and more modules deposited by other projects. [ISOD 1992]

There are other examples of research that address the retrieval problems, for example, retrieval of reusable software by behavior sampling [PROD 1993], and source code retrieval using program patterns [PAUL 1992]. If more software artifacts are found in the repository, the effort
of software reuse activity becomes smaller and savings in the software development also become larger.

1.2 Objectives

In this dissertation, we
1) survey literature on different aspects of software reuse to understand the underlying philosophy of software reusability,
2) survey historical and current trends of software reusability to assess advantages and disadvantages of each discipline,
3) observe the software development process as the basis for software reusability,
4) define and describe features that are necessary for a software reusability system, but that do not currently exist in a single reusability model, and
5) design a model that includes the necessary features in order to support software reuse of all phases in the software development and maintenance process.

* * * * *

In Chapter 2, we survey the definition, a brief history, reasons, objects, and types of software reusability.
2. LITERATURE REVIEW

2.1 Definition of Reusability

Reusability is an activity of using something again [FREE 1987a]; however, in the context of software engineering, software reuse is the process of using existing software artifacts during the construction of new software. [KRUE 1992] This definition is based on four assumptions. First, existing software artifacts can be reused. Second, reuse of existing software artifacts gives benefits in comparison with software development from scratch. Third, reuse of existing software artifacts requires different types of processes. The process involves the selection, specialization, and integration of artifacts [KRUE 1992]. Last, different types of software artifacts can be reused. The types of artifacts include source code fragments, design structures, module-level implementation structures, specifications, documentation, and transformations. [KRUE 1992]

Biggerstaff and Perlis [BIGG 1989] define Software Reuse as follows:

Software reuse is the reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of that other system. This reused knowledge includes artifacts such as domain knowledge, development experience, design decisions, architectural structures,
requirements, designs, code, documentation, and so forth.

This definition is more expansive in three aspects. First, the reuse activity is not simply a reuse but a reapplication. Second, reuse is applied to maintenance as well as development. Third, in addition to the types of artifacts listed above, Biggerstaff and Perlis have suggested the reuse of knowledge including domain knowledge, development experience, design decisions, and architectural structures. [BIGG 1989] Knowledge plays an extremely important role in software engineering because the productivity of a software developer and the quality of the software produced are dependent to a large extent on the experience and skills of the individual. [LEE 1993] Although the above definition is expansive, it may be difficult to realize the reapplication of knowledge in a rigorous way in terms of software engineering, for example, development experience or design decisions. Some knowledge is lost when it is not captured explicitly or embedded within a software artifact. Therefore, the knowledge must be encoded in a software artifact as software concepts in order to be reused. [LEE 1993]

Tracz added an important aspect to the general definition of software reuse. "Software reuse, ⋯, is the process of reusing software that was designed to be reused."
G. Jones also suggests that a way to minimize the risk of failure is to carry out development in such a way that the effort can be used in some future projects as well as the current one. A part of the current problem in software reusability is due to the fact that the great majority of the existing software was not designed to be reused. Current software methodologies and procedures do not include reuse as part of their process. So, the way software is developed has to be changed to support software reuse. 

We emphasize four points regarding the definition of software reusability. First, software reuse is the activity of reusing all types of software artifacts. The types of artifacts to be reused will be discussed in detail later in this chapter. Second, the software reuse process involves logical and physical tools, i.e., mechanisms and computer tools of software selection, adaptation, integration, classification, standardization, etc. Generally, an integrated computer-aided software engineering (CASE) technology is believed to provide considerable support for software reuse. Third, the concept of software reuse must be introduced at early stages of the software development. Fourth, software must be designed to be reused.
2.2 Brief History of Reusability

The idea and practice of software reuse is not new. From the early days of computer programming, programmers have spontaneously reused the code of frequently required routines or blocks of code stored in their code libraries, with or without modifications. However, the number of reusable routines was small, the propagation of the routines was limited, and theories that supported the methodology of reuse so that it could be an effective factor in reducing the software development effort were not yet mature.

The realization of software reuse started with the origins of the stored program computer EDSAC at the University of Cambridge in 1949. [TRAC 1988a]

In the 1968 NATO Software Engineering Conference, the birthplace of software engineering [KRUE 1992], Dough McIlroy introduced the concept of formal software reuse. [PRIE 1993] McIlroy proposed a library of reusable software components, automated techniques for customizing components to different degrees of precision and robustness [KRUE 1992], and standardized catalogues of routines. [MCIL 1968] His idea was key to the software factory concept.

In the early 1970s, technical developments in various areas, i.e., languages, data structures, operating systems, program transformations, and specification techniques, practiced reuse at the code level but, the concept of reuse
was not yet broadly employed. Developments in design representations and the methods for creating them, for example, SADT, structured design, program design languages, Jackson design, or Warnier/Orr diagrams, can be considered as the reuse of design but, the concept of reuse was not explicit, either. [FREE 1987a]

In 1974, Systems Development Corp. (SDC) registered the term "software factory," used in the report of McIlroy, as a trademark. SDC established a well-defined set of procedures for a consistent and repeatable software process but, software reuse was still implicit because reuse was not defined in the process. [PRIE 1993]

In the mid-1970's, Raytheon Company's Missile Systems Division began a reuse project led by Robert Lanergan. [1983] This project is considered the first formal reuse effort at the organization level. [PRIE 1993] In terms of software reusability, the period before 1978 was a prehistoric period, containing only unconscious developments and primitive applications of reuse. [FREE 1987a] Interest in reuse spread through academia in the late-1970s.

The proceedings of the First International Workshop on Reusability in Programming, held in 1983, was the best collection of papers and reports of several research projects in the early 1980s. In this workshop, reuse at several levels of abstraction was discussed. [PRIE 1993] In
1984, Ted Biggerstaff and Alan Perlis summarized and classified the trends of research reviewing the issues discussed in the 1983 Workshop. Generally in the 1980s, significant contributions of the creation of large-scale reuse programs were realized by the US Advanced Research Projects Agency, Eureka Software Factory, Hitachi, Toshiba, NEC, Fujitsu, NTT, GTE, AT&T, IBM, Hewlett-Packard [PRIE 1993], DEC, BTG Inc., U.S. Army, Raytheon Company, and Hartford Insurance [INCO 1990].

In the late-1980s, several advances were made in library systems, classification techniques, the creation and distribution of reusable components, reuse support environments, and corporate reuse programs. However, reuse did not deliver its promise of significant improvement in productivity and quality of software. [PRIE 1993] In 1988, Basili and Rombach extended the definition of reuse to products, processes and knowledge. [BASI 1988]

Recent work has addressed nontechnical issues including management, economics, culture, and law. Management issues address upper management emphasis on software reuse, training on reuse concept, and incentives on reuse either via promotion or via funding. [INCO 1990]

Cultural issues are to some extent related to management issues. The idea is to advocate software reuse to software developers in order to cultivate software reuse
as a self-motivated development culture. In successful reuse programs, such as BTG Inc., software reuse is simply the way software is developed. [INCO 1990]

As we pointed out earlier, the idea of software reuse came from the effort to enhance the productivity and the quality of software. The other critical aspect of the software reusability is the reduction of the cost. Margono and Rhoads [MARG 1992] emphasized the economics of software reuse as follows:

In order to be successful, not only must a reuse program be technically sound, it must also be economically worthwhile.

From this sense, the consideration of economics in software reuse is a prerequisite to software reuse practice. A large body of literature about economics issues exists, including reuse metrics [PFLE 1990], cost/benefit tracking methods [MARG 1992], reuse investment/benefit analysis [BARN 1991], and relative cost/productive formula [BARN 1988]. One successful practice is the Advanced Automation System (AAS) of the United States Federal Aviation Administration (FAA). [MARG 1992]

Law issues include copyright and technique transfer. Since current software reuse activities tend to be local to each organization, the legal aspects are not a critical factor; however, for the inter-organizational and,
ultimately, global realization of software reuse, legal problems must be considered.

These nontechnical problems are important but difficult to solve. The view of reuse today is to integrate all these factors into the idea of institutionalized reuse. [PRIE 1993]

2.3 Reasons for Reusability

The primary motivation to reuse software artifacts is to reduce the time and effort required to build software systems. [KRUE 1992] In other words, productivity can be enhanced and the cost may be reduced by using well-tested software components. G. Jones discussed, "Productivity can be increased by developing software that will provide the same functionality and require the production of fewer delivered source instructions. This is done by: reusing existing software: …." [JONEGW 1990] Furthermore, reliability can be achieved [BOLD 1989] because reusable software artifacts are already well-tested and used, possibly maintained, in the real world.

Reuse can be applied to the maintenance process. [KRUE 1992] The debugging costs can be amortized and larger complex system can be constructed easier. [BOLD 1989]

Tracz summarized the major reasons for reusing software: productivity and quality. [TRAC 1988a] Software reuse increases productivity because programmers may write
fewer lines of code when large portions of code or design are copied verbatim, the amount of documentation and testing is reduced, the system becomes easier to maintain and modify due to the familiarity with the reusable building blocks and design, and frequently-used components can be migrated into microcode, special hardware, or silicon. Estimated productivity variation between the best and worst performance by software engineers is of the order of 5:1 to 10:1. However, in specific circumstances, the variation becomes 16:1 to 28:1. For instance, averaging over the lifetime of a project, individuals may be found to produce anything between 3 and 50 lines of code per day, showing a 16:1 productivity ratio. [JONEGW 1990] Regarding that striking ratio, the performance of the least productive software engineers can be improved by reusing products of the most productive software engineers. Another aspect is that improvement in quality can be achieved because reusable software components are well designed, well documented, well tested, and easily understandable. [TRAC 1988a]

Surveys on various programs have shown the results that encourage software reusability with following statistic:

- 60% of the all business application designs and code can be standardized and reused, [LANE 1983]
- 40% - 60% of all code can be standardized and reused, [LANE 1983]
75% of program functions are common to more than one program, [JONE 1983]

only 15% of the code found in most programs is unique, novel, and specific to individual applications, [JONE 1983] and

over 60% of all functions in commercial business applications falls on four types of operations: print report, file maintenance, file build/load/copy, and posting/updating. [GOOD 1983]

Also, there are a number of reports about successful reuse practices. Some of these reports include:

Raytheon Missile Systems Division reported an average 60% reused code in their new systems and a net 50% increase in productivity over a period of six years [LANE 1983],

NEC's Software Engineering Laboratory in Tokyo reported 6.7 fold productivity improvement and 2.8 fold quality improvement by reusing 32 standardized logic templates and 130 common algorithms [LANE 1983],

Fujitsu reported 20% of 300 projects were on schedule before reuse was applied, but after Software Development for Electronic Switching Systems (SEDSS) Program, a simple and pragmatic
reuse approach, was introduced, the number increased to 70% [JONE 1983],

- GTE reported 14% reuse in the first year saving $1.5 million by utilizing the Asset Management Program (AMP). Their projected figures for 1995 are 50% reuse and $10 million in savings [PRIE 1991b],

- In NTT, after a four year experiment, the reuse ratio (reused module size/developed program size * 100) in the fourth year increased to 20% and is continuously increasing [ISOD 1992],

- BTG Inc. claims that systems contain 50% to 80% reused components. Also, reuse compressed software development schedules by factor of three to five [INCO 1990],

- In both The First Boston Corporation (FBC) and Carter Hawley Hale Information Services (CHH), reuse percentages were about 35% [BANK 1993].

2.4 Object of Reuse

Freeman discussed two observations that lead to a broad definition of a reusability object. First, the economic motivation for reusability is to reduce the labor component that is necessary to produce a new software system. Second, programming is only a small portion of the cost of creating a system. So, the object of reusability is any information
which a developer may need in the process of creating software. This definition is too broad to motivate specific research. Therefore, he narrowed the object in a hierarchy of reuse types. [FREE 1983] The reuse types are discussed in the next section.

Boldyreff's definition of the reuse object is broader. "The history of ideas, intellectual progress provides a paradigm for reusability. Here progress is effected by developing and refining the ideas of others." [BOLD 1989] The ideas of others consist of theory, which is agreed principles, and praxis, which is accepted practice. The established principles and proven practice would be combined with formal and informal methods. Literature [documentation of programs] is one medium for passing ideas and ensures reuse of software. Reuse of common code sequences with the subroutine library concept is the engineering foundation of software construction. Design principles, in the form of abstraction, would be reused. Specification gives a clear statement of the theory and concept underlying the software. All these components are useful to generalize, not only the constituent parts of a system, but also its structure by inductive reasoning over a number of similar systems. [BOLD 1989]
Basili and Rombach indicate that reuse of products, processes, and knowledge will be the key to an effective reuse technology. [BASI 1988]

In general, there is agreement from two perspectives on what to reuse. First, reusability must be achieved at different abstraction levels. Reuse of code is recommended but reuse of code alone is not effective to improve productivity and quality of software. So, all types of products including programs, documents, experience, and knowledge must be reused. Second, reuse must be established as an explicit concept in the software engineering domain. Reusability must be a principle of the development process, instead of a side effect of the development of software.

2.5 Software Reuse Process

Different authors address different software reuse processes [BIGG 1987] [BOLD 1989] [REDW 1989] [GALL 1992], but they agree generally upon five activities: decomposition and abstraction, cataloguing and classification, selection, adaptation, and composition.

The first step of software reuse starts with decomposition and abstraction of existing software, or software being developed. A large software system must be decomposed into component concepts. [BOLD 1989] Decomposition involves abstraction so that implementation detail is concealed. Instead, at this stage, the
functionality of the component to be decomposed must be the only consideration of the component concept.

Each component must be described and categorized. Description of components involves cataloguing that translates the concepts of components into a predefined language. Catalogued components are categorized with respect to the description. A common method to categorize components is the classification of the concepts according to some schema. [BOLD 1989] Catalogued and classified components are stored into a component library. Library maintenance and retrieval mechanism are required for efficient and effective use of the library. [REDW 1989]

Software components stored in a components library will be selected and retrieved for reuse. The selection process involves identifying potentially applicable components through either query-based search or browsing. [REDW 1989] Search mechanism takes an important role in selection in terms of efficiency. Selected components will be retrieved for reuse.

The selected and retrieved components may be modified to meet the new application's requirements. This process is the adaptation. Adaptation also involves customization, tuning, or extension to meet the needs of the current project. [REDW 1989]
Once a component is retrieved and adapted, it must be composed into a full system context. Composition involves editing and linking the component. [REDW 1989]

The five software reuse processes discussed above may be categorized largely into two activities. One is the activity for reuse. The other is the activity to reuse. While the former is simply a preparation for future reuse, the latter is an actual process of software reuse. Decomposition/abstraction and cataloguing/classification are activities for reuse. To reuse activities include location, understanding, adaptation, and composition of components.

For reuse activities and to reuse activities are not stand-alone activities. They are closely related. For instance, the decomposition and abstraction processes are strongly coupled with understanding in the to reuse process. The selection process is also largely defined by cataloguing and classification process, for example, search strategies.

2.6 Trends of Software Reusability

From the technology viewpoint, reusability can be divided into two major groups according to the characteristics of components being used: composition technologies and generation technologies. Composition approaches emphasize either application component libraries or organization and composition principles of building blocks. Generation approaches fall in one of three
categories: language based generators, application generators, and transformation systems. Generation-based systems take at least two different forms of patterns: patterns of code and patterns within transformation rules. In both cases, the effect of the individual reusable components within the target program tends to be more global and diffuse than the effects of building blocks. [BIGG 1987]

Neither of these strategies are totally successful. They reveal fundamental drawbacks as well as advantages. In composition technologies, the components to be reused are atomic and immutable, but passive. This technology requires solutions to the following problems: mechanisms for identifying useful components, methods for specifying components, the form of components, and method for cataloging components. [HORO 1989] In generation technologies, formalism and its application to the development of a large system have to be successfully addressed. On the other hand, even if the problems involved in the use of formalism are successfully solved, a question arises: Why don't we reuse existing software written in formal languages? Then, generation technologies may face the same fundamental problems raised by composition technologies.
2.7 Types of Software Reusability

In this section, we will discuss different types of reuse methodology generally being practiced with their advantages and disadvantages.

Horowitz and Munson reviewed successful examples of software reusability: subroutine libraries, compilers, simulation, and parameterized systems. [HORO 1989] In a broad sense, all four examples can be considered as reusability methods, but they do not attract today's researchers in this area.

Subroutine libraries have two forms. [HORO 1989] One is the library of functions or procedures provided as general utilities. The other form is the library of functional elements that can be applied to an application area. Neither of the two forms of subroutine libraries can be adapted, i.e., modification of algorithm is not allowed. Documents provided with the library were, at most, manuals of its use, but they were not documents to be adapted to the new environment, i.e., new software.

Compilers transform a sequence of source code to another sequence of target code, but transformation rules enforce exactly the same skeleton of code patterns. Modification of target code is not user-friendly after compilation. No documentation is generated about target code, either.
**Simulation** has for a long time been recognized as a major type of generic application for computers. These concepts have been incorporated into programming languages such as GPSS (an extension of Fortran) or Simula (an extension of ALGOL-60). These are the extension of general-purpose programming languages with simulation concepts. Essential elements common to all simulations are defined using the extended programming languages. These elements can be reused for a new application instead of being recreated. [HORO 1989] However, simulations are not be considered as an actual realization of software.

A **parameterized system** is a single all-encompassing system for a specific domain. A software developer can select options from a parameterized system using a questionnaire for a specific application. The result of the questionnaire is transformed into a preprocessing program. This preprocessor specializes the large system for the application. However, a single parameterized system lacks of flexibility and control in order to address variety of systems in different domains. [HORO 1989]

**High-level programming languages** can be considered as a way of reuse. The common implementation patterns used in assembly language programming became the primitive constructs in high-level programming languages. Some of the most impressive productive gains of the last 40 years have
come from the use of higher-level programming languages, which have allowed software developers to get more effect from less code and to free themselves to many of the nonessential details of their work. [JONEGW 1990] Reusable assembly language patterns provide a factor of 5 speedup in development time because high-level programming language constructs are succinct and natural, compilers are fully automated, and programmers are isolated from compilers and assembly language details. However, it is not widely recognized that high-level programming languages are examples of software reuse that scientists and software developers are interested in nowadays. [KRUE 1992]

The notion of reusable code has attracted software developers and researchers since the advocacy of McIlroy's component manufacturing facility because of its simple concept. Software creation in the large can be achieved by constructing the building blocks from the components factory through the mechanical design of families of software components that are written, tested, and stored, like "structural shapes, screws or resistors," [MCIL 1968]. Criticism followed about this concept on four points. [HORO 1989] [KRUE 1992] First, creation of a parameterized system that is efficient, reliable, and convenient for all systems is impossible. Second, machine dependencies can not be avoided. Third, it is very hard to create index terms for
cataloging components in such a way that anyone can search for the proper component. Fourth, source code is difficult to understand. Ideally, the well-tested components are used in the new software development or maintenance. However, the effort of locating, understanding, modifying, and debugging a component may exceed the effort of creating the equivalent component from scratch.

Software schemas, or templates, are a formal extension to reusable software components, but emphasis is on reusing abstract algorithms and data structures rather than source code. Later, abstraction specification for a schema can be automated to the realization of source code. When schemas are based on formal specifications, automated tools can be used for selection, specialization, and integration. Software developers work at a higher level of abstraction. However, formal specifications for schema abstraction can be large and complex. Even with automated tools, it may be difficult to locate, understand, and use schemas. [KRUE 1992]

The idea of very high-level program-producing systems is to create a processor that interacts with a software developer in order to produce executable code. This approach may represent a continuation of the high-level programming languages trend, but they are generally non-procedual so that software developers just define the
problem by allowing them to concentrate on specifying what is to be done, rather than how. [JONEGW 1990] Some approaches including reusable design, reusable processor, transformation systems, application generators, and prototyping followed this model. [HORO 1989] Advantages and drawbacks will be examined with the discussions of each methodology.

The key idea of reusable design is to study a particular application domain in a formal way and the artifacts of this study are used to design software systems that automate the domain. The domain analysis can be reused for any types of system in the same domain. Two questions arise: the types of domain analysis output, and the method to manipulate output. Domain analysis output types include data types, processes, algorithms, and constraints, etc. Here, the question is how these outputs are recorded for later use. Domain analysis output can be manipulated in two ways. The first way is to translate data types and processes into code modules written in a programming language. The second way is to extend existing programming languages to Simula for simulation with the new types and processes. [HORO 1989] If a software developer is able to adapt the design of a large system, software development cost will be relieved largely regarding the design takes 70% to 90% of a software system. [ARAN 1993] However, in the
worst case, a software developer may spend more time understanding, modifying, and translating the design into executable programs. [KRUE 1992] Misunderstanding of a design may highly escalate the software cost because lack of understanding may account for 50% to 90% of design cost. [ARAN 1993] Rather than reusing a reusable design, a **reusable processor** can be built for a single, sophisticated specification language. The language is non-procedural. In the specification, the order of operations is not required. Rather, developers specify only operations to be built. The specification is translated into executable code automatically. Subsequent debugging can be done in the original source. Although the specification language is difficult to understand, software developers need not learn a new language. Quality documentation is produced along with the code. [HORO 1989] High-level abstraction can be achieved with this approach.

**Transformation systems** take high-level operational specifications in order to transform them into efficient programs. Debugging is done at the specification level. Therefore, the specification language must be sufficiently concise and descriptive, in order for the specification be readable and understandable; furthermore, the specification itself is the documentation. [HORO 1989] Transformation
systems use general-purpose, high-level programming abstraction. The abstraction can be at a higher level because the software developers can guide transformations to produce more efficient implementations. However, applying transformations may require time and effort from the software developers. [KRUE 1992]

The original purpose of application generators was to generate programs whose output is a series of reports. Later, they were extended to interface with an existing database, to perform statistical operations, and to report the results. The programmer writes statements provided by the application generator to produce an executable program that accesses the database and reports the result. This approach requires a non-procedural language for a non-technical users. Also, application generators must have built-in knowledge about an existing database management system, or about the specification of data files on the computer it is running on.

Application generators address several advantages. First, application generators are non-procedural. Programmers can concentrate on what to do to solve the problem, instead of how to solve it. Second, they enable programmers to create arbitrary applications from data files. Third, application generators have built-in knowledge about concerning file creation, record input and
output, report writing, and various logical operations on fields of records. Last and most importantly, debugging can be avoided because the generated code has already been debugged. Modifications of the program are made in the non-procedural language level. [HORO 1989] Database management procedures are frequently accessible to end-users and represent a process that is often called "automatic programming." [JONEGW 1990] However, because of the limited availability, many of application generators have narrow domain coverage. It is often difficult, or impossible to find an application generator for a particular application. Also, it is very hard to build an application generator with appropriate functionality and performance for a broad range of applications. [KRUE 1992]

Although prototyping is too expensive, it is powerful for defining the requirements of a computerized system. Hands-on use of a prototype is an ideal way to determine if the projected system performs in the desired way. Two facets of research are recommended. The first facet is how can it be inserted into the software life cycle effectively. The second facet is how prototyping can be more economical. Three forms of research were suggested: study of new prototyping languages, the development of prototypes into efficient systems, and incremental modification of a prototype into a production system. LISP and APL have
proved to be most useful in creating prototypes, but they are inefficient for production systems. [HORO 1989] Other languages, like PROLOG, SETL, and RAPID/USE seemed to be particularly handy during the early stages of prototyping. [JONEGW 1990]

Programming language support is an important factor in software reusability because a programming language influences programmers in the way they think in order to solve their problems through the concept it offer. [HORO 1989] Ada and object-oriented programming languages successfully advocate the concept of software reuse.

The Ada programming language offers several mechanisms in the specification and development of reusable software. The Package concept is a mechanism to define an abstract data type that supports information hiding. The generic procedure enables programmers typing of a procedure which performs the same operation over different data types. [HORO 1989] The package construct supports modularization and separation between specification and implementation. [TRAC 1988c] The strong typing enforces consistency between formal and actual parameters. [TRAC 1988c] Finally, overload resolution provides some syntactic and semantic reuse of function. [TRAC 1988c]

The notion of inheritance from the object-oriented programming languages offers a unique contribution to the
software component reuse. Class definitions are data abstraction mechanism and subclasses are specialization of their super classes. [KRUE 1992] Encapsulation mechanism binds data and operations on it so that a class can be treated as a unit of reuse.

Very high-level languages are an attempt to improve and to extend the conventional high-level languages. Emphases are on the specification via constructs. Executable code can be obtained automatically through specifications. A research goal is to maximize the leverage offered by higher levels of specification without losing computational generality. [KRUE 1992] Examples of popular very high-level programming languages are Focus, Oracle, Mantis, Ramis II, dBase [JONEGW 1990], SETL, PAISLEY, and MODEL [KRUE 1992]. Very high-level languages are more succinct, natural, and expressive than high-level programming languages. High-level mathematical abstractions are automatically transformed into executable codes. Very high-level languages are suitable for general-purpose programming. However, the run-time performance is very poor. Use of specification based on mathematical abstractions tends to make average software developers difficult to understand.

Software architectures represent a large-scale global structure of a software design that can be used as a whole. This approach is analogous to very large-scale software
schemas. However, while very large-scale schemas focus on data structures and algorithms, software architectures focus on subsystems and their interaction. Software architectures are also analogous to application generators in the sense that large-scale system design is reused. Application generators are typically standalone systems with implicit architectures, while software architectures are explicitly specialized and integrated with other architectures to produce different composite architectures. Reusable architectures can be used either stand-alone to create end-user applications, or as building blocks to create higher-level software architecture. However, creating reusable software architecture is difficult and the application domain is narrow. [KRUE 1992]

2.8 Summary

Incorvaia and Davis [INCO 1990] concluded their case studies on software reuse practices as follows:

Software reuse is occurring in industry today, albeit informally. Perhaps the most important observation is that there is no right or wrong way to practice software reuse. ... It seems more important to find an approach that fits the culture than to follow some textbook method.

As we surveyed in sections 2.6 and 2.7 above, different reuse methodologies have different advantages and disadvantages. An advantage of one methodology can be a disadvantage of the other, and vice versa. However, we
believe that there are issues that must be considered to realize more efficient and more effective software reuse. In order to achieve more successful software reuse, i.e., enhance productivity and quality of software economically, we focus on the following aspects:

1) Software must be designed to be reused,
2) Reuse effort must be partly absorbed by development activities, except the amortization of software cost by virtue of software reuse, without imposing an intensive investment,
3) A reuse methodology must support all phases of software development life cycle,
4) A reuse methodology must be independent from software development methodologies, and
5) A reuse methodology must support the adaptation of existing software into the new software.
3. A MODEL FOR REUSABLE SOFTWARE

From the literature as discussed in chapter two, the need for improved reusability models is clearly evident. In addition, criteria that such a model should possess are identified and emphasized by the current researchers in software reuse. The criteria include:

1) Software must be designed to be reused,
2) A software reusability system must be independent from software development methodology,
3) A software reusability system must support all phases of software development process,
4) A software reusability system must support software maintenance as well as software development,
5) A software reusability system must support adaptation of existing software to a new environment,
6) A software reuse activity must require neither an intensive initial investment nor an excessive effort, and
7) A software reusability system must easily be applied to the standards of each organization.

In this chapter, we present a model that meets this criteria. First, we define terminology and introduce
notations used in the text and diagrams. Then, we give some observations related to the software development life cycle and reuse-oriented software development. Finally, we present the Quintet Web model and each of five Webs in detail.

Throughout this chapter, we use a waterfall model which includes user specification, requirement specification, design, pseudo-code, code, and testing phases to show relationships between phases of the software life cycle. However, the Quintet Web may be customized for different software development environments. For instance, the design phase may be divided into two phases: functional design and detailed design. In this case, the waterfall model consists of seven phases, instead of six. The Quintet Web can be adapted to the seven phases. Once the Quintet Web is adapted to a specific environment for a user group, and is utilized during the software development, a predefined phases and documents standards set by the user group will be enforced. This will encourage the standardization of software development methodology and environment.

3.1 Definition of Global Terminologies

We use terminologies defined in this section globally throughout this dissertation. Other terminologies that are used in local sections are defined locally.
A phase is a stage in the software development life cycle. User specification, requirement specification, design, pseudo-code, code, and test each constitutes a phase.

An adjacent phase is the phase immediately prior to or immediately following to a phase.

A document refers to any type of machine readable products from any phase of the software development process. It includes specifications, design documents, pseudo-code, code, and test data.

Decomposition (decompose) is the breaking down of a word which has a broad semantic meaning into more than one word, each of which has a narrower semantic meaning than the original word. Usually, as decomposition is carried on, the abstraction level goes down, i.e., more detailed issues are introduced. When decomposition is applied to a document, or a part of a document, the document is split into more than one sub-document, usually, according to their functionalities.

A keyword is any term that represents a concept of functionality in a software system or its environment definition. For example, a term that describes the operation of a code segment is a keyword.

A sub-keyword is a keyword that has a narrower semantic meaning than a keyword. The abstraction level of a sub-
keyword is less than that of a keyword. If a keyword \( W \) can be decomposed into three keywords \( W_1, W_2, \) and \( W_3 \) each of which has a narrower semantic meaning of \( W \), then \( W_1, W_2, \) and \( W_3 \) are each a sub-keyword. This term is used relative to a keyword.

A **block** designates a part of any document that is decomposed as an entity. A block is associated with one or more keywords that describe the operation of the block. Many authors use this terminology as a component, a segment or an artifact in order to designate a similar concept, but we use the term block because it best suits the concept in our context. In a specification, a word, sentence, paragraph, or the whole specification can be embedded in a block. In code, one or a group of instructions, or one or more functions and/or procedures can be a block.

A **sub-block** is a block that is a block that has been defined as a result of a decomposition. For example, if a block \( B \) is decomposed into three blocks \( B_1, B_2, \) and \( B_3 \), then \( B_1, B_2, \) and \( B_3 \) are each sub-blocks.

A **link** is a logical connection between two blocks in a document or in two different types of documents. When this term is used in contrast to the term "batch link (defined below)," the term "single link" will be used. For example, the link between a block in the user specification and a block in the requirement specification is a (single) link.
A **batch link** is a collection of single links *between two phases*. For instance, links of two blocks in the user specification to their corresponding blocks in the requirement specification form a batch link.

A **chain** is a collection of single or batch links *among more than two phases*. For example, a block in the user specification is linked to its corresponding block in the requirement specification, and the block in the requirement specification is linked to the corresponding block in the design. These two links among the three phases are called a chain. Even when the links among the phases are batch links, the term "chain" is used. However, when a chain consisting of batch links must be distinguished from a chain that consists of single links, we will use the term a **"batch chain"** to designate the chain of batch links.

A **web** consists of all links between different phases or blocks in a phase at various detail levels and in various types of documents, i.e., multiple links of the same type. It may be represented by batch links, for brevity.

A **layer** is links of blocks between two phases at a given level of abstraction. All levels of links form a multiple-layered link construct.

A **dependency tree** is a tree in which the operation of every block (except the root block) is dependent on its parent block. For instance, if a block $B$ calls blocks $B_1$, $B_2$, $B_3$, and $B_4$, then $B$ is the root block, and $B_1$, $B_2$, $B_3$, and $B_4$ are its child blocks.
$B_2, \ldots, B_n$, then $B$ becomes the root with children $B_1, B_2, \ldots, B_n$. Recursively, if there exist dependencies between $B_1$ and its sub-blocks $B_{11}, B_{12}, \ldots, B_{1m}$, then $B_1$ becomes the root of the sub-tree, and its sub-blocks $B_{11}, B_{12}, \ldots, B_{1m}$ become children of $B_1$.

3.2 Definition of Global Notations

In this section, we define notations that are used globally in this dissertation. Other notations that are used locally in specific sections are defined in the local sections.

$U, R, D, T, C, T$: Unless specified otherwise, these letters will be used to represent a user specification, a requirement specification, a design, a pseudo-code, a code, or a test phase and their associative documents, respectively.

**Link**: A single lined bidirectional arrow ($\leftrightarrow$ or $\uparrow$) designates a link. We use this notation in the diagrams generally if there is no confusion in distinguishing different types of links.

**Batch Link**: A double lined bidirectional arrow ($\leftrightarrow$ or $\uparrow$) represents a batch link. Batch links simplify either of two instances: one instance is an implication of a relationship between two phases of the software development processes or between two blocks in each phase. The other instance is an involvement of multiple Webs in a
relationship between two phases where the identification of the Webs is not important in the context. If a keyword \( W \), consisting of two sub-keywords \( W_1 \) and \( W_2 \), has a batch link to a document block \( B \), consisting of two sub-blocks \( B_1 \) and \( B_2 \) such that \( W_1 \) is linked to \( B_1 \), and \( W_2 \) is linked to \( B_2 \). A higher level batch link \( W \leftrightarrow B \) can be formed. However, in this batch link, only \( W_1 \leftrightarrow B_1 \) and \( W_2 \leftrightarrow B_2 \) can be included. Links \( W_1 \leftrightarrow B_2 \) and \( W_1 \leftrightarrow B_2 \) must not be included.

**Block:** A pair of brackets ("[" and matching "]") represents that any contents between them forms a block. For instance, \([B_1, B_2, \cdots, B_n]\) is a block consisting of \( n \) sub-blocks \( B_1, B_2, \cdots, B_n \). A block is distinguished from a set. In a set, the order of elements in the set is not important but, the order of elements in a block cannot be changed arbitrarily. This requirement is because the behavior of a sequence of software components may be changed if the sequence is changed. For example, \( \{B_1, B_2, B_3\} = \{B_2, B_1, B_3\} \), but \( [B_1, B_2, B_3] \neq [B_2, B_1, B_3] \). If there is no confusion, a block consisting of a single sub-block is represented as \( B \) instead of \([B]\), for simplicity.
Figure 3.1  Definitions

Figure 3.1 illustrates most terminologies and notations defined above.

3.3 Observations

In order to show the relationship between the phases in traditional and reuse-oriented software development life cycle, we redefine the notation \( \rightarrow \) as a relation, and introduce a new relation \( \| \). We also introduce mapping functions \( \mathcal{E}, \mathcal{E}^r, \mathcal{P}, \text{ and } \mathcal{P}^r \).

\( \rightarrow \) is a notation to represent a 'move' of the Turing Machine, or 'yield in one step' of a Pushdown Automaton.
Here, we redefine this notation as the evolutilional relation as follows:

\[ A \rightarrow B: \text{ } B \text{ is developed or created to realize } A \text{ in the later phase of software development life cycle, where} \]
\[ A \text{ and } B \text{ are blocks of software developed in different phases.} \]

In the traditional software development environment, relationships \( U \rightarrow R \rightarrow D \rightarrow P \rightarrow C \) hold.

A mapping function \( \mathcal{E} \) is defined as follows:

\[ \mathcal{E}: u \rightarrow v, \text{ where} \]
\[ u \text{ is a software block developed in a phase, and} \]
\[ v \text{ is one or a set of corresponding blocks developed in the next phase.} \]

Using the relation \( \rightarrow \) and the mapping function \( \mathcal{E} \), we can formulate the evolutilional relationship of all phases in the software development life cycle as follows:

\[ \forall v \in V \left( \exists w \in W \left[ (v \rightarrow w) \land (\mathcal{E}(v) \rightarrow w) \right] \right), \text{ where} \]
\[ v, w: \text{ blocks developed in some phase, and} \]
\[ V, W: \text{ phases of the software life cycle.} \]

The evolutilional relationships of all phases in the traditional software development life cycle can be formulated as follows:

\[ \forall w_u \in U \left( \exists w_c \in R \left[ (w_u \rightarrow w_c) \land (\mathcal{E}(w_u) \rightarrow w_c) \right] \right), \]
\[ \forall w_r \in R \left( \exists F \in D \left[ (w_r \rightarrow F) \land (\mathcal{E}(w_r) \rightarrow F) \right] \right), \]
∀ F ∈ D [∃ [p] ∈ T [ (F \models [p]) \land (\mathcal{E}(F) \rightarrow [p]) ]], and
∀ [p] ∈ T [∃ [c] ∈ C [ ([p] \models [c]) \land (\mathcal{E}([p]) \rightarrow [c]) ]],

where

\( W_u \) : one or a set of operational keywords in the user specification,
\( W_r \) : one or a set of operational keywords in the requirement specification,
\( F \) : an operational function in the design,
\( [p] \) : a block of pseudo-code, and
\( [c] \) : a block of code.

Artifacts of the previous phase evolve to the artifacts of the next phase. However, the relationship between the code and test phases is different because a block of code does not evolve to a set of test data. Instead, a set of test data is produced to test the validity of a block of code. This relationship is formulated employing another mapping function \( \mathcal{P} \).

A mapping function \( \mathcal{P} \) is defined as follows:

\( \mathcal{P} : p \rightarrow q \), where

\( p \) is an artifact from a phase of the software life cycle, and
\( q \) is information that must be produced to validate a block of code \( p \).

Consequently, the relationship between the code and testing phases can be expressed as follows:

∀ [c] ∈ C [∃ (t) ∈ T [\mathcal{P}([c]) \rightarrow (t) ]], where

\( [c] \) : a block of code,
\( C \): entire code,
\( \{t\} \): a set of test data, and
\( T \): entire test data.

As we observed above, in the traditional software development process, software evolves from one phase to the next phase. But, little explicit information about the evolution between the phases is maintained. For example, the link between a part of the requirement specification and the corresponding part of the design is not kept. If we preserve the information, the location of a segment of software to be reused would lead to the reuse of all phases associated with the segment. Software is best known to the developers when it is being developed. So, if information that can be used when the software needs to be reused is created and preserved, the reuse effort can be expected to be reduced.

Before the evolution information is utilized, at least one segment of software must be located. A keyword search leads software developers to locate a segment of software. Semantic goals of each part of the software can be translated into a set of keywords. These keywords can be used in the search. Information of the link between the keywords and segments of software in each phase must be kept. We call these two types of information the Reuse Support Information (RSI). For example, a link from \( u \) to \( v \) to show that \( u \) has evolved to \( v \) is a type of RSI. A
link from a keyword \( W \) to a block of software \( [b] \) is another type of RSI.

In order to represent reuse-oriented software development concepts, we introduce an evolitional relation with RSI \( \vdash \) as follows:

\[
A \vdash B: \ B \text{ is developed with a set of RSI to realize } A \text{ in the later phase.}
\]

Analogously, in the reuse-oriented software development environment, relationships \( U \vdash R \vdash D \vdash T \vdash C \) hold.

A mapping function \( \mathcal{E}^{R} \) is defined as follows:

\[
\mathcal{E}^{R}: u \rightarrow v \times i, \text{ where } u \text{ and } v \text{ are the same as in the definition of } \mathcal{E}, \text{ and } i \text{ is a set of RSI.}
\]

With \( \vdash \) and \( \mathcal{E}^{R} \), the software development life cycle that creates RSI can be formulated as follows:

\[
\forall v \in V \ [ \exists w \in W \ [ v \vdash w \land (\mathcal{E}^{R}(v) \rightarrow w + \text{RSI}) ] ].
\]

The evolitional relationships of all phases in the reuse-oriented software development life cycle are formulated as follows:

\[
\forall W_{u} \in U \ [ \exists W_{l} \in R \ [ (W_{u} \vdash W_{l}) \land (\mathcal{E}^{R}(W_{u}) \rightarrow W_{l} + \text{RSI}) ] ],
\]

\[
\forall W_{l} \in R \ [ \exists F \in D \ [ (W_{l} \vdash F) \land (\mathcal{E}^{R}(W_{l}) \rightarrow F + \text{RSI}) ] ].
\]
∀ F ∈ D [∃ [p] ∈ P ([F ∈ [p]] ∧ (G^R(F) → [p] + RSI)]), and

∀ [p] ∈ P [∃ [c] ∈ C ([[p] → [c]] ∧ (G^R([p]) → [c] + RSI)]),

where

\( W_u, W_r, F, [p], \) and \([c] \), are same as the definition in the traditional software development life cycle, and RSI is Reuse Support Information.

For the same reason as explained above, to show the relationship between the code and test phases, we need to define a mapping function \( P^R \) as follows:

\( P^R: p \rightarrow q \times i, \) where

\( p, q \) are same as in the definition of \( P \), and \( i \) is a set of RSI.

So, the relationship between coding and testing phases is formulated as follows:

\( ∀ [c] ∈ C [∃ (t) ∈ T [P^R([c]) → (t) + RSI] ], \) where

\([c], C, \) \((t)\), and \( T \) are same as in \( P \), and RSI is reuse support information.

As we observed above, for each step of the software development process, RSI can be created for future reuse of the software currently being developed.

In the following sections, we describe a multi-faceted reusability model, the types of RSI that should be
created for the model, and how the RSI is utilized in the reuse of software.

3.4 Quintet Web: A Model for Reusable Software

The Quintet Web consists of five Webs: Semantic Web, Horizontal Web, Vertical Web, Syntactic Web, and Alternative Web. Figure 3.2 conceptually illustrates the Quintet Web.

![Quintet Web Diagram]

**Figure 3.2** A Quintet Web: an Abstract View

Thick radial lines represent the five Webs. The center of the radial lines, labeled $S$, is an intersection of the five Webs. A user of the Quintet Web begins at $S$. From $S$, 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
he can expand his search to any direction: to the Semantic Web, to the Horizontal Web, to the Vertical Web, to the Syntactic Web, or to the Alternative Web. The five thick arrows pointing outward from the center $S$ symbolize the search activities of reusable blocks by different strategies. If a user, following a hierarchical structure of a system, investigates procedures or functions in the hierarchy, he will expand his search through the Vertical Web. This activity is illustrated by the arrow labeled H. If the user decided to trace semantics, he will traverse the Semantic Web directed by the arrow labeled K. Arrow P represents the search activity through the Horizontal Web specifying a phase of development life cycle. Arrow V depicts a variable search activity through the Syntactic Web and arrow B portrays an investigation of alternative blocks through the Alternative Web. Solid circles on each Web are arbitrary locations in each Web. Each circle is linked to all other Webs via bidirectional links. A user can access any Web at any location in the Quintet Web through the links. So, if a user moves from $S$ to $S'$, $S'$ becomes the center of the radial lines, i.e. Webs. Spiral lines at $S'$ become radial lines; however, no physical change is made because this change is logical, like a spider moving to another location of its virtually infinitely large orb web preserves possibilities to move to any direction from the
new location. At $S'$, the user can expand his search to any direction. A portion of the Vertical Web is enlarged to show that there is more than one spiral line in the Quintet Web. $S''$ is another possible center of the Quintet Web. There may be many potential centers between $S'$ and $S''$.

3.4.1 Semantic Web

A keyword that represents the functionality in a system may be realized by a group of particles scattered through the whole system. In other words, every statement in the specifications, or every instruction in the pseudo-code or code has its own semantic goal(s). A group of statements or instructions, not necessarily a consecutive sequence, may have the same semantic goal. They are grouped together through a link to show how or by which pieces of the system the goal was realized. We call this set of links a Semantic Web.

The Semantic Web is links of blocks in each of the user specification, requirement specification, pseudo-code, and code according to the semantic meaning of each block.

In a system, this link logically starts from a keyword or a combination of keywords using logical operators. If a block has more than one semantic meaning, the block would be linked in more than one link. The Semantic Web may consist of several layers. For instance, a keyword initiates a link to blocks whose semantic goals are described by the keyword.
However, the semantic meaning of the keyword is broad enough to be decomposed to a set of keywords with narrower semantic meanings so that they together can construct the original semantic meaning of the broad keyword. Each set must have a Semantic Web, but the union of blocks linked in each Semantic Web is equivalent to the Semantic Web of the broad keyword. Formally, we describe the concept of layers as follows:

Let a keyword \( W = \{ W_1, W_2, \cdots, W_m \} \), where \( m \) is a positive integer and a sub-keyword \( W_i, 1 \leq i \leq m \), is unique, be a set of keywords in a document of any phase. Then, each \( W_i \) is a keyword that has a semantic meaning. \( W_i \) may be decomposed to a set of keywords \( \{ W_{i1}, W_{i2}, \cdots, W_{in} \} \), where \( n \) is a positive integer and each keyword \( W_{ij}, 1 \leq i \leq m, 1 \leq j \leq n \), is unique, each of which has a narrower semantic meaning than \( W_i \). Then a set of keywords \( W \) becomes:

\[
W = \{ \{ W_{i1}, W_{i2}, \cdots, W_{in} \}, \{ W_{21}, W_{22}, \cdots, W_{2n} \}, \cdots, \\
\{ W_{m1}, W_{m2}, \cdots, W_{mn} \} \} ,
\]

where \( n_k \) is a positive integer and \( 1 \leq k \leq m \).

Let a block \( B = \{ B_1, B_2, \cdots, B_m \} \), where \( m \) is a positive integer and a sub-block \( B_i, 1 \leq i \leq m \), is unique, be a set of document blocks in any phase document. Let \( B_i \) be a block associated with \( W_i \), i.e., \( B_i \) is a block
that realizes $W_i$. Again, $B_i$ may be decomposed to a set of blocks $(B_{i1}, B_{i2}, \cdots, B_{in})$, where $n$ is a positive integer and each $B_{ij}, 1 \leq i \leq m, 1 \leq j \leq n$, is unique such that each $B_{ij}$ is associating block of each keyword $W_{ij}$.

Analogously, a set of block $B$ becomes:

$$B = \{ \{B_{11}, B_{12}, \cdots, B_{1n}\}, \{B_{21}, B_{22}, \cdots, B_{2n}\}, \cdots, \{B_{m1}, B_{m2}, \cdots, B_{mn}\}\},$$

where $n_k$ is a positive integer and $1 \leq k \leq m$.

The Semantic Web that a set of keyword $W$ initiates can be represented as follows:

$$W \leftrightarrow B,$$

where

- $W$ is a set of keywords,
- $B$ is a set of software blocks, and
- $\leftrightarrow$ is a bidirectional batch link.

For $W = (W_1, W_2, \cdots, W_n)$ and $B = (B_1, B_2, \cdots, B_n)$, $W \leftrightarrow B$ is transformed as follows:

$$(W_1, W_2, \cdots, W_n) \leftrightarrow (B_1, B_2, \cdots, B_n).$$

Again, this batch link is transformed to a set of batch links as follows:

$$(W_1 \leftrightarrow B_1, W_2 \leftrightarrow B_2, \cdots, W_n \leftrightarrow B_n).$$

However, if all $W_i$ and $B_i, 1 \leq i \leq m$, can be decomposed no more, the transform becomes:

$$(W_1 \leftrightarrow B_1, W_2 \leftrightarrow B_2, \cdots, W_n \leftrightarrow B_n).$$
Since $W_i = \{W_{i1}, W_{i2}, \ldots, W_{in}\}$ and $B_i = \{B_{i1}, B_{i2}, \ldots, B_{in}\}$, this set of links becomes:

\[
\{ \{W_{i1} \leftrightarrow B_{i1}, W_{i2} \leftrightarrow B_{i2}, \ldots, W_{in} \leftrightarrow B_{in}\},
\{W_{j1} \leftrightarrow B_{j1}, W_{j2} \leftrightarrow B_{j2}, \ldots, W_{jn} \leftrightarrow B_{jn}\},
\ldots
\{W_{m1} \leftrightarrow B_{m1}, W_{m2} \leftrightarrow B_{m2}, \ldots, W_{mn} \leftrightarrow B_{mn}\} \}.
\]

If all $W_{ij}$ and $B_{ij}$, $1 \leq i \leq m$, $1 \leq j \leq m$, are the lowest layer sub-keywords and sub-blocks, respectively, all batch links ($\leftrightarrow$) used in above set of links become single links ($\leftrightarrow$). This means that if no further decomposition is possible for both sub-keywords and sub-blocks, they are linked (not batch linked) to form the lowest layer of the Semantic Web. The lowest layer is formed by a set of links, for example,

\[
\{ \{W_{i1} \leftrightarrow B_{i1}, W_{i2} \leftrightarrow B_{i2}, \ldots, W_{in} \leftrightarrow B_{in}\},
\{W_{j1} \leftrightarrow B_{j1}, W_{j2} \leftrightarrow B_{j2}, \ldots, W_{jn} \leftrightarrow B_{jn}\},
\ldots
\{W_{m1} \leftrightarrow B_{m1}, W_{m2} \leftrightarrow B_{m2}, \ldots, W_{mn} \leftrightarrow B_{mn}\} \}.
\]

However, there may be a situation where a sub-keyword cannot be decomposed, but, in order to achieve a better modularization, the sub-block that realizes the sub-keyword has to be decomposed into multiple sub-blocks. For instance, $W_{ij}$ is a sub-keyword that cannot
be decomposed, but $B_{ij}$ is decomposed into multiple sub-blocks $[B_{ij1}, B_{ij2}, \ldots, B_{ijk}]$, $1 \leq i \leq m$, $1 \leq i \leq n$, $k \geq 2$. In this case, the lowest layer $W_{ij}$ and $B_{ij}$ construct becomes:

$$W_{ij} \leftrightarrow [B_{ij1}, B_{ij2}, \ldots, B_{ijk}].$$

As a summary, the Semantic Web is a set of links that links keywords and software blocks bidirectionally. It consists of different layers according to the abstraction level.

Semantically, the equations

$$W = \bigcup_{i=1}^{m} W_{i} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} W_{ij} = \bigcup_{i=1}^{m} \bigcup_{j=1}^{n} \bigcup_{k=1}^{l} W_{ijk} = \cdots$$

and

$$B = \bigcup_{i=1}^{m} B_{i} = \bigcup_{i=1}^{m} \bigcup_{j=1}^{m} B_{ij} = \bigcup_{i=1}^{m} \bigcup_{j=1}^{n} \bigcup_{k=1}^{l} B_{ijk} = \cdots$$

must always be preserved. Some block $B_{ij}$ may be the same as the block represented as $B_{i'j'}$, where $i \neq i'$, $j \neq j'$, if a block has more than one semantic meaning. For each keyword, starting from the highest to the lowest, layers can be constructed recursively.
Figure 3.3  Layers in the Semantic Web

The decomposition just described is necessary to provide a user with semantic information of fragments of a system from the broadest level to the narrowest level. Figure 3.3 illustrates the concept of layers in the Semantic Web.

A keyword $W$ is considered at the highest layer. $W$ is batch linked to a block $B$ which is also at the highest layer. The keyword $W$ is decomposed into $m \geq 2$ sub-keywords $W_i$ through $W_n$. The block $B$ is also decomposed into $m \geq 2$ sub-blocks $B_i$ through $B_m$. Each $W_i$, $2 \leq i \leq n$, is batch linked to the
associated $B_i$. Again, a sub-keyword $W_i$ and the corresponding sub-block $B_i$ are decomposed into $n \geq 2$ sub-keywords $W_{i1}$ through $W_{in}$ and into $n \geq 2$ sub-blocks, respectively. Each of the $W_{ij}$ and the corresponding $B_{ij}$ must also be batch linked. However, while a sub-keyword $W_{ij}$ cannot further be decomposed, its counterpart $B_{ij}$ is decomposed into $k \geq 2$ sub-blocks $B_{ij1}$ through $B_{ijn}$. So, $W_{ij}$ must be linked to a block of sub-blocks $[B_{ij1}, B_{ij2}, \ldots, B_{ijn}]$. Sub-keywords $W_{i2}$ through $W_{in}$ are linked to their corresponding sub-blocks $B_{i2}$ through $B_{in}$.

In this diagram, the highest layer is:

$$W \leftrightarrow B.$$ 

The next highest layer is:

$$\{W_1 \leftrightarrow B_1, W_2 \leftrightarrow B_2, \ldots, W_m \leftrightarrow B_m\}.$$ 

The lowest layer link is as follows:

$$\{W_{i1} \leftrightarrow [B_{i11}, B_{i12}, \ldots, B_{i1n}], W_{i2} \leftrightarrow B_{i2}, \ldots, W_{in} \leftrightarrow B_{in}\}.$$ 

Figure 3.4 is an example of the Semantic Web. $W_1$ through $W_5$ are keywords. $U_i$'s, $R_i$'s, $D_i$'s, $P_i$'s, and $C_i$'s, are the $i$th block of user specification, requirement specification, design, pseudo-code, and code, respectively. Upright arrows (↑) are pointers to the blocks.

Figure 3.5 shows links from keywords to each corresponding blocks of the user specification and requirement specification.
Figure 3.4  Semantic Web

Figure 3.5  Links from Keywords to User Specification and Requirement Specification Phases

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Keywords are linked to pseudo-code and code in the same manner. Figure 3.6 shows the links from keywords to corresponding blocks of pseudo-code and code.

![Diagram](image)

**Figure 3.6 Links from Keywords to Pseudo-code and Code Phases**

Links from keywords to design are shown in Figure 3.7. Dotted closed curves show blocks. The set of links in Figure 3.7 is as follows:

\[
\{ \begin{array}{l}
W_3 \leftrightarrow \{ D_3, D_{31} \}, W_4 \leftrightarrow \{ D_4, D_{41}, D_{43} \}, \\
W_5 \leftrightarrow \{ D_5, D_{51} \}, W_6 \leftrightarrow \{ D_6 \} 
\end{array} \}
\]

The sets of links for Figure 3.5 and Figure 3.6 are obvious, so they are not shown here.
3.4.2 Horizontal Web

The Horizontal Web links the blocks of software from the different phases. A specification may be decomposed into one or more blocks according to each operation so that each block addresses an individual operation. A block in the user specification may be reconstructed as one or more corresponding blocks in the requirement specification. A block of the requirement specification may be transformed to a group of blocks in the design phase. Similarly, a block of the design can be realized as a group of pseudo-code, or code blocks. However, each block of code has a set of corresponding test data.
Figure 3.8 is an example of a Horizontal Web. The boundary drawn by a thick curved line represents the complete set of documents in the design phase. Each phase (except the user specification and test data being linked only to the right or left, respectively) is connected by two links to its right and left adjacent phases. These links are the top level links. When a designer wishes to retrieve other phase documents of the corresponding phase in which he is currently investigating, the top level links are used. However, if the designer wants to retrieve blocks of other
phase documents that correspond to the blocks he is dealing with, links that connect blocks, i.e., lower links, are used.

The Horizontal Web consists of links from blocks of each phase to one or more blocks of other phases. However, the Horizontal Web is by no means a linear link. It must be constructed as multiple layers of link levels as in the Semantic Web.

The Horizontal Web and the Semantic Web are closely related. First, documents of all phases are conceptually decomposed in order to construct layers of blocks. Then, the Semantic Web links all blocks to their operational concepts, i.e., corresponding keywords. According to the level of decomposition, multiple layers are constructed. The Horizontal Web uses the decomposition formed by the Semantic Web in order to link blocks of other phases. Therefore, the idea of the layer is identical. For example, a small block in the requirement specification has a link to a block in the design which is represented as a structure chart. Then, the larger block in the requirement specification that includes the small block will be linked to the parent block that includes the small one as its child in the structure chart.
The concept of multiple layer in the Horizontal Web is illustrated in Figure 3.9. Similar links would be built between user specification and requirement specification, design and pseudo-code, pseudo-code and code. However, the direct links of one phase to non-adjacent phases are not required unless the adjacent phase document is not available. For example, a direct link between the user specification and the design is not recommend, because a non-adjacent phase of the user specification, i.e., design, can be traced through the links of adjacent phases, in this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
case, links between user specification and requirement specification, then requirement specification and design.

When an adjacent phase document is not available, a link must jump to the next-adjacent phase, but, there must be a comment that the document of the phase is not available. If a block of one phase can not be linked to its previous phase, we consider the previous phase incomplete. A modification or update is needed to complete the previous phase. If the previous phase is considered to be complete, but the current phase document cannot be linked, the block of the current phase is either redundant or the level of abstraction is too low to be discussed in the previous phase. For the former case, there are two choices. One choice is to include the operation currently being developed to all previous phases in order to make the operation legitimate. The other choice is to remove the operation being developed for the current phase. For the latter case, a message to the user must be augmented to explain the situation that the level of abstraction of this block is too low to be included in the previous phase.

This feature of the Horizontal Web enables software developers to help ensure the validity of software being developed or maintained. The consistency of software from the user specification through code must be maintained. The software inconsistency consists of three aspects:
redundantly added operations, operations that are specified but not implemented, and different features of the specification realized and the implementation.

Redundant portions of software increase software cost in two ways. First, lines of code along with other phases documents will be increased and more test effort is required. Naturally, software development cost will be higher. Second, the volume of software to be maintained will be larger. Considering that the ratio of the maintenance cost to the development cost is 3:1 [HORO 1989], the cost of software cost will be escalated.

Missing operations, i.e., operations specified in specifications documents but not implemented, may cause problems in the later phases, or in the worst case, after the software development is complete. Unexpected perfective maintenance will be a costly extra expense.

If the behavior of software does not match its specifications, a corrective maintenance must be accomplished. The specification or design phase are likely to be involved causing a very expensive retreat.

However, in the Horizontal Web, a block of a phase may not be directly linked to the matching block of its adjacent phase. This is because the level of abstraction in decomposition is not able to be the same between phases. For example, a block of the user specification \( \mathcal{U} \) is realized
as a large block of the requirement specification $R$ which includes three sub-blocks, $R_1$, $R_2$, and $R_3$. There is a link between $U$ and $R$, but there is no link from $R_1$, $R_2$, or $R_3$ to any block in $U$. If a user wants to find a matching block of $R_i$ in $U$, he must trace up to $R$ and trace the link between $R$ and $U$. A part of the block $U$ may match with $R_3$. This link is an indirect link, because, in order to trace the link, a user must trace up through the higher block of the phase. Even though $R_1$, $R_2$, and $R_3$ are not directly linked to $U$ in this example, an indirect link is not considered as an inconsistency or illegitimate.

There are two types of links in the Horizontal Web: direct and indirect links. The direct link is the link from a block to its corresponding block(s) of the adjacent phases. The indirect link is not an actual link, but the link that can be traced through the other phase or through a higher block in which a sub-block resides. There are two types of indirect links: through-the-phase link and through-the-layer link. A through-the-phase link is a virtual link of a block to the block of its non-adjacent phase through its adjacent phase(s). A through-the-layer link is a virtual link of a block to its adjacent phase through its larger block where it resides, or the lower sub-block it contains.
Figure 3.10 Direct and Indirect Links in the Horizontal Web

Figure 3.10 represents direct and indirect links. Solid arrows are direct links. These links are actual. Dotted arrows and dashed arrows are indirect links. They are virtual links, i.e., there are no actual links between blocks, instead, blocks are connected via indirect links.

In the diagram, dashed arrows are through-the-phase link. Block \( U_i \) in the user specification does not have a direct link to block \( D_i \) in the design, but, because \( U_i \) has a link to \( R_i \), and \( R_i \) has a link to \( D_i \), \( U_i \) can be considered to have a link to \( D_i \) indirectly, and vice versa. In a
similar manner, block \( U \) is indirectly linked to \( D \) via links \( U \) to \( R \), and \( R \) to \( D \). A transitive relation holds in the through-the-phase link.

Dotted arrows are through-the-layer links. A sub-block \( R \) in requirement specification does not have a link to the user specification and the design, but, \( R \) is linked to \( R \) where it resides. \( R \) has links to \( U \) and \( D \). So, \( R \) can be linked to \( U \) or \( D \) through \( R \) indirectly, and vice versa.

There are obvious similarities between the Semantic Web and the Horizontal Web. Links in both webs are constructed through the semantics of blocks. However, the Horizontal Web identifies the associated blocks in different documents with a block whereas the Semantic Web identifies a group of blocks that are associated with a single or a combination of keywords in each phase. These two webs may be implemented as a single web, but conceptually, they are distinct. Briefly, a Horizontal Web is a set of inter-document links and a Semantic Web is a set of intra-document links. Keywords are explicit in the Semantic Web and implicit in the Horizontal Web.

Every system is implemented by code written in some programming language. However, software reuse is essentially different from the reuse of code. [KARA 1989] Moreover, as Karakostas mentioned, "Reusing software at the
early stages of its development, i.e. when it is still in the form of specifications, or designs, is an intuitively appealing approach, mainly because of its big added value and its relative independence from implementation details."[KARA 1989] From this respect, the Horizontal Web is essential in terms of software reusability, because it enables us to make the most of labor-intensive, time-consuming, error-prone activities, i.e. reuse of specifications and designs incorporate a significant added value.

Formally, the Horizontal Web is given below.

Definition:

\[ B_i^g: \text{the } i^{th} \text{ block in the phase } S \]
\[ B_{ij}^g: \text{the } j^{th} \text{ sub-block of the } i^{th} \text{ block in the phase } S \]

The highest layer of the Horizontal Web consists of a chain of six elements and five batch links as follows:

\[ B_1^u \leftrightarrow B_2^u \leftrightarrow B_3^u \leftrightarrow B_4^u \leftrightarrow B_5^u \leftrightarrow B_1^u. \]

Since a block in a phase may be realized as multiple blocks in the next phase, each batch link can be unravelled to a set of single links. The batch link

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
of the first two blocks \((B_f^1 \leftrightarrow B_f^2)\) is transformed as follows:

\[
\{ B_f^1 \leftrightarrow B_f^1, B_f^2 \leftrightarrow B_f^2, \ldots, B_f^1 \leftrightarrow B_f^2 \},
\]

provided that \(B_f^2\) is decomposed into \(n \geq 1\) sub-blocks.

For simplicity, we transform this set as follows:

\[
\{ B_f^2 \leftrightarrow [B_f^1, B_f^2, \ldots, B_f^n] \).
\]

The other batch links can be unravelled in the same manner except \(B_f^2 \leftrightarrow B_f^2\). Resulting batch links are:

\[
\{ B_f^2 \leftrightarrow [B_f^1, B_f^2, \ldots, B_f^n] \),
\]

\[
\{ B_f^2 \leftrightarrow [B_f^1, B_f^2, \ldots, B_f^n] \), and
\]

\[
\{ B_f^2 \leftrightarrow [B_f^1, B_f^2, \ldots, B_f^n] \).
\]

\(B_f^2 \leftrightarrow B_f^2\) is unravelled differently. Test data is decomposed according to the decomposition of code. Therefore, every sub-block of \(B_f^2\) must be linked to only one sub-block of \(B_f^2\). However, the concept of layers in \(B_f^2 \leftrightarrow B_f^2\) persists because a code block at the highest layer may be linked to a set of test data blocks so that the code block can be reused along with its test data.

3.4.3 Vertical Web

A Vertical Web has conceptual similarity to structure charts, Warnier-Orr diagrams, or Jackson diagrams. When a function or a procedure call is made in a subprogram, the caller and callee have a bidirectional link so that all the ancestors and descendants of the subprogram located by a user can be traced through the links. Inheritance in the
object-oriented design/programming may be represented as links in a similar manner. In other words, the Vertical Web identifies the dependency of a block of software to another block. So, it is not only for a design that involves hierarchical structure, but also for object-oriented design or other design methods where a dependency-like relationship exists, i.e., caller-callee, client-server, sender-receiver, or class relationship.

If a software developer finds a block of software to reuse for a new project or for the maintenance of existing software, he may need to investigate the relationship of the block with other blocks it depends on or by which it is dependent on. The software developer can investigate the callers and the callees through the Vertical Web. If the callers and/or the callees comply with the software developer's needs, he can expand the block to a larger block including the block he found. Callers or callees may be selectively chosen.

Generally, the larger the block, the more time and effort in software development or maintenance will be saved. However, there is a trade-off. If a software developer traces up through a Vertical Web and finds that he could reuse a group of blocks, more cost savings would be achieved, but the group of blocks is less probable to be frequently reused. Conversely, if a software developer
traces down to a leaf, the cost savings achieved may be smaller, but the probability of the block being frequently reused is higher [BIGG 1987]. When the effort to be imposed for an adaptation is insignificant, or the cost of the adaptation is less than that of development, a group of hierarchical blocks (but not necessarily a hierarchical blocks) will be reused. If the cost of an adaptation exceeds that of development, the software developer will search for the better match to his needs, or he will develop the portion of the system. However, for efficient and effective reuse, the knowledge of one subprogram's environment and the relationship between the subprogram and the environment are important factors.

In addition, the system design process is not typically a pure top-down process. Rather, it is a combination of top-down and bottom-up steps. [LENZ 1987] The Vertical Web provides these aspects. The diagram in Figure 3.11 is a Vertical Web. The array on the top consists of five pointers to corresponding documents: the first pointer points to the user specification, the second points to the requirement specification, the third points to the design, the fourth points to the pseudo-code, and the fifth pointer points to the code. More precisely, each pointer points to the highest block in the hierarchy of each document. This link is unidirectional. From/to the highest block from/to
the next highest blocks, until the lowest blocks are linked, bidirectional links develop a complete Web of each document. These bidirectional links enable a software developer to investigate the hierarchical relationship between blocks easily. Note that blocks 5, 6, 9, and 10 in design, pseudo-code, and code are realized after the requirement specification phase.

Figure 3.11 Vertical Web

Attributes of each block are inherited to its sub-blocks unless attributes of a sub-block are modified by its own attributes. An example of the inheritance of attributes...
is illustrated in Figure 3.12. Each sub-block is represented as a box with label $B_n$, $1 \leq n \leq 6$. Attributes are represented as pointed boxes with labels $\text{ATTR}_m$, $m = 1, 3, 5$. Dashed attributes are inherited attributes. In the diagram, a sub-block $B_1$ is characterized by its attribute $\text{ATTR}_1$. Attributes $\text{ATTR}_1$ and $\text{ATTR}_3$ are attached to sub-blocks $B_3$ and $B_5$ to characterize the corresponding sub-blocks $B_3$ and $B_5$. $\text{ATTR}_1$ is entirely inherited to sub-blocks $B_2$ and $B_4$. However, for sub-blocks $B_3$ and $B_5$, their corresponding attributes $\text{ATTR}_3$ and $\text{ATTR}_5$ take precedence over the inherited attributes $\text{ATTR}_1$. Descendants of $B_1$ may be partially inherited $\text{ATTR}_1$ if their own attributes are not fully characterize the sub-blocks. For example, if $\text{ATTR}_5$ partially defines $B_5$, a sub-block $B_5$ inherits $\text{ATTR}_1$ partially, i.e., $B_5$ inherits attributes not defined by $\text{ATTR}_5$ but defined by $\text{ATTR}_1$. Sub-blocks without attributes, i.e., $B_2$, $B_4$, and $B_6$ must inherit attributes. The sub-block $B_2$ inherits attributes from $B_1$. So, the sub-block $B_2$ is characterized by the inherited attributes $\text{ATTR}_1$. $B_4$ inherits $\text{ATTR}_1$ from $B_1$. However, a sub-block $B_6$ inherits $\text{ATTR}_3$ instead of $\text{ATTR}_1$ because $B_1$ is closer than $B_1$. If the sub-block $B_3$ partially inherits from $B_1$, $B_6$ also partially inherits from $B_1$ attributes that are not defined in $\text{ATTR}_1$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 3.12  Inheritance of Attributes in the Vertical Web

Formally, the Vertical Web is defined below:

Let $V$ be the Vertical Web and $[b]$ be a block of software that is the root of the dependency tree. If $[b]$ has no child, then the Vertical Web of that software is an empty set:

$$V = \emptyset.$$  

If $[b]$ has only one child $[b_1]$, then the Vertical Web consists of a single link:
If \([b]\) has more than one child \([b_1], [b_2], \ldots, [b_n]\), \(n \geq 2\), then the Vertical Web consists of a set of links:
\[
V = \{ [b] \leftrightarrow [b_1], [b] \leftrightarrow [b_2], \ldots, [b] \leftrightarrow [b_n] \}.
\]
For simplicity, we use a batch notation to denote the same set as the above as follows:
\[
V = \{ [b] \leftrightarrow [[b_1], [b_2], \ldots, [b_n]] \}.
\]
In general, the Vertical Web \(V\) is of the form:
\[
V = \{ [b] \leftrightarrow [[b_i] \leftrightarrow [[b_{ij}] \leftrightarrow [[b_{ijk}] \leftrightarrow \cdots]]] \},
\]
where \(0 \leq i, j, k, \ldots,\) and \(i, j, k, \ldots\) are integers.

Differences of link type between the Vertical Web and the Horizontal Web are obvious. In the Vertical Web, all blocks are hierarchically linked within a phase but, in the Horizontal Web, links are constructed between blocks of different phases. Also, in the Horizontal Web, the unit of a block relies on the functionality represented by a keyword, but the unit of a block in the Vertical Web depends upon a hierarchical context.

### 3.4.4 Syntactic Web

If we believe in the importance of the knowledge of program structure in program debugging, software maintenance, or software reuse, then the syntactic structure
of a program may be the most fundamental knowledge. We present this type of information in the Syntactic Web. The Syntactic Web shows the syntactic chain of variables within and beyond a block. We call these two types of chain the intra-component link and the inter-component block.

An **intra-component link** is a link for every variable that occurs in the block, starting from its declaration (or the first occurrence of each variable when implicit declaration is allowed and used) to every location it occurs. All the variables in a block can be traced through the intra-component link.

An **inter-component link** is a link of every variable whose scope is beyond the scope of the block in which it resides, starting from the declaration of the variable to the places it is used. The use of global variables or variables whose scope is beyond the block can be traced through the inter-component link. Figure 3.13 is a simple example of a Syntactic Web. Dashed arrows represent the intra-component links and solid arrows represent the inter-component links.
A formal definition of the Syntactic Web is given below.

Definition:

\( \nu, \omega: \) variables

\( p: \) a parameter

\( \mathcal{F}, \mathcal{G}, \mathcal{H}, \mathcal{I}: \) programs, functions, or procedures

\( \nu(\mathcal{F}_a): \) a variable \( \nu \) in an actual parameter list of \( \mathcal{F} \)
\(v(I)\): a parameter \(v\) in a formal parameter list of \(I\)

\(v(F)\): a variable \(v\) declared in \(F\)

\(v^i(F)\): the \(i^{th}\) occurrence of a variable \(v\) in \(F\), \(i \geq 1\)

\(v(F)\): all occurrences of a variable \(v\) in \(F\).

Five possible cases for a variable to be declared and/or used in a block of code are discussed below:

1. If \(v\) is declared but not used anywhere, then the Syntactic Web is linked to an empty set:
   \[
   \{ v(F) \leftrightarrow \emptyset \}. 
   \]

2. If \(v\) is a local variable, then the Syntactic Web consists of a set of links:
   \[
   \{ v(F) \leftrightarrow v^1(F), v(F) \leftrightarrow v^2(F), \\
   \cdots v(F) \leftrightarrow v^n(F) \}, n \geq 1. 
   \]
   Simply, this set can be denoted as:
   \[
   \{ v(F) \leftrightarrow \{v^1(F), v^2(F), \cdots, v^n(F)\} \}, n \geq 1. 
   \]

3. Let \(v\) be a global variable declared in the main program \(I\). Let \(G, H,\) and \(I\) are procedures or
functions in which \( v \) is used. The Syntactic Web consists of a set of links:

\[
\{ v(\mathcal{F}) \leftrightarrow v^1(\mathcal{G}), v(\mathcal{F}) \leftrightarrow v^2(\mathcal{G}), \ldots, \\
v(\mathcal{F}) \leftrightarrow v^l(\mathcal{I}), v(\mathcal{F}) \leftrightarrow v^m(\mathcal{J}), \ldots, \\
v(\mathcal{F}) \leftrightarrow v^n(\mathcal{K}) \}.
\]

This set is simplified as follows:

\[
\{ v(\mathcal{F}) \leftrightarrow \{ v(\mathcal{G}) \}, v(\mathcal{F}) \leftrightarrow \{ v(\mathcal{J}) \}, \\
\ldots, v(\mathcal{F}) \leftrightarrow \{ v(\mathcal{I}) \} \}.
\]

4. Let \( v \) be a variable used in a procedure (or a function) \( \mathcal{F} \) as an actual parameter for a procedure (or a function) call \( \mathcal{G} \). \( w \) is a variable used in \( \mathcal{G} \) as a formal parameter. Then the Syntactic Web consists of a set of chains:

\[
\{ v(\mathcal{F}) \leftrightarrow v(\mathcal{G}_a) \leftrightarrow w(\mathcal{G}_b) \leftrightarrow w^1(\mathcal{G}), \\
v(\mathcal{F}) \leftrightarrow v(\mathcal{G}_a) \leftrightarrow w(\mathcal{G}_b) \leftrightarrow w^2(\mathcal{G}), \\
\ldots, \\
v(\mathcal{F}) \leftrightarrow v(\mathcal{G}_a) \leftrightarrow w(\mathcal{G}_b) \leftrightarrow w^n(\mathcal{G}) \}, n \geq 1.
\]
For simplicity, we represent this set of chains as follows:

\[
\begin{align*}
\{ v(F) \leftrightarrow v(G_a) \leftrightarrow w(G_t) \leftrightarrow \\
[w^1(G), w^2(G), \ldots, w^n(G)] \}\.
\end{align*}
\]

Again, this set is simplified as follows:

\[
\{ v(F) \leftrightarrow v(G_a) \leftrightarrow w(G_t) \leftrightarrow \{w(G)\} \}.
\]

5. If \( v \) is a variable declared implicitly, the first occurrence of the variable is considered as its declaration. Since the variable \( v \) is not declared but is used, case 1 discussed above does not occur. For cases 2 through 4, each \( v(F) \) is replaced with \( v^i(F) \). The rest of the formulation remains the same.

The ideal situation is to reuse the blocks that are stored in a database without modification, but it is overly optimistic to expect that we can build a reusability system that allows significant reuse without the need to modify any portion of the blocks. Furthermore, the concept of modification changes the perception of a reusability system from a static library of building blocks to a living system of components that spawn, change, and evolve new components with the changing requirements of their environment. [BIGG 1987] The Syntactic Web is helpful especially when blocks

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
in a reusability system need to be adapted to a new environment, or to be edited to comply with different documentation standards.

3.4.5 Alternative Web

The Alternative Web consists of a set of links. A block of one system is linked to a block of the other system(s) if the blocks perform the identical operation. If there is more than one block in the reusability system that performs the same operation but in different ways, all of them are linked together to provide a user with a choice of methods to attain the same functionality.

A block may merely give an alternative to other blocks. That is, the block is independent and is not a part of a larger system. We call the block an orphan alternative because it has neither predecessor nor descendant. Of course, this block must have other phase documents. For example, if a software developer locates a code block, he must have access to corresponding blocks in the specifications, pseudo-code, and test data through Horizontal Web.

On the other hand, a block which is linked to other blocks through an Alternative Web may be a sub-block of a larger system. This may not be unusual, because more than one system may have procedures or functions with the same functionality. If a reusability system includes more than
one system, each system may have alternatives of some blocks in other systems. When a software developer finds an alternative block in another system, he may want to investigate the design of a portion of the other system around the block. This type of block may be a clue to the other system's technology so that a software developer can refer to the design of another system, either a part or all of it, through the other system's Quintet Web. We call this block a **bridge alternative** because this block provides a link from one system to another, a portion of which implements the identical operation. The Alternative Web is not necessarily a link of single blocks. It can be a link of groups of blocks each of which has a vertical relationship, if each group accomplishes the same functionality. The scope of the Alternative Web is each development phase, but not between phases. (The relationship between phases is addressed in the Horizontal Web.)

Each block or group of blocks has an **attribute tag** that consists of attributes such as the design methods, programming language and its version, time complexity, space complexity, algorithm, special requirements, and specifications, to provide a software developer with a quick reference before he proceeds. Each phase involves different types of information according to its characteristics; for
instance, an attribute tag of a block in a user specification may not include time complexity, but the one in a design or a code does.

Figure 3.14 Alternative Web

The Alternative Web is shown in Figure 3.14. There are two independent systems, $S$ and $T$, and an independent block in the reusability system. Solid arrows connect blocks in each system as alternatives of each other. So, they are called bridge alternatives of each system. However, dashed arrows connect blocks in a system to the blocks which are not a part of any system, i.e., they are independent blocks.
The purpose of the independent blocks is to provide alternatives to other blocks. So, they are called orphan alternatives. The dotted arrows are links that connect blocks and their attribute tags, or blocks to other blocks. When a software developer investigates an alternative of the block found, the information included in the attribute tags of all candidate blocks is given to the software developer in order to provide the description of each alternative. This link is also bidirectional to provide a mechanism to locate the actual document that is chosen.

Formally, the Alternative Web is defined below.

Let $S$ and $S'$ are software systems such that $S \neq S'$. \{ $U, R, D, T, C, T$ \} $\in S$, and \{ $U', R', D', T'$, $C', T'$ \} $\in S'$ are sets of phases from the use of the waterfall model. $B$ is a block of any phase document. $B^k$ is a block developed in a phase $K \in \{ U, R, D, T, C, T, U', R', D', T', C', T \}$.

\[
( \exists_{B \in S, B' \in S'} ( B \leftrightarrow B' ) ) \rightarrow \\
((B'^u \leftrightarrow B'^u) \land (B^k \leftrightarrow B^k) \land \\
(B^o \leftrightarrow B'^o) \land (B^e \leftrightarrow B^e) \land \\
(B^c \leftrightarrow B^c) \land ((B^c \leftrightarrow B^c) \land (B^c \leftrightarrow B^o)))
\]

If $B'$ is an orphan alternative, $S'$ may be a component library that consists of more than one (possibly many)
components. The component library also includes modularized software blocks along with their other phase document blocks that are maintained with the Quintet Web.

3.4.6 Summary

These five types of webs that fabricate a Quintet Web coexist. All links in the Quintet Web are bidirectional to provide a user with the ability to investigate blocks bidirectionally, and to change webs at any time and at any place in the Quintet Web.

The Semantic Web is composed of links of blocks from keywords that describe the respective blocks. The scope is limited to specifications, design, pseudo-code, and code. The test phase is not considered to have semantic goals of its own, because test data are strongly bound to the code they verify. This web is multi-layered. A keyword initiates a link to blocks whose semantic goals are identified by the keyword itself. The semantic meaning of the keyword can be decomposed to multiple keywords with narrower but more specific meanings. Each of the sub-keywords initiates a link, as well as the primary keyword. In other words, corresponding webs are developed according to the degree of abstraction of a keyword.

The Horizontal Web consists of links of blocks in different phase documents. The scope of this link is all phases. Each phase is decomposed to blocks, and each block
has links to blocks in other phases. This is a multi-layered link, thus, each decomposed block has its own layer of links. There are two types of links: direct link and indirect link. A direct link is the link from a block to a block of its adjacent phase. Otherwise, a link is indirect. An indirect link consists of two types of links: through-the-phase link and through-the-layer link. A through-the-phase link is the link of a block to its non-adjacent phases. A through-the-layer link is the link of a block to the blocks in a different layer. The Horizontal Web enables a software developer to reuse early stage software knowledge prior to the consideration of implementation details.

The **Vertical Web** consists of links of blocks by their operational dependencies. This is a single layer web. The Vertical Web enables a software developer to decide the hierarchical level of reusable software. More importantly, this link provides the flexibility for both top-down and bottom-up approaches, or a combination. This web may be represented as a dependency tree. The root block is not dependent on any block. However, all other blocks are operationally dependent on their parent blocks.

The **Syntactic Web** is made up of links of variables in a system. The scope of this link is mainly for the code phase. Two types of links are in this web: inter-component link and intra-component link. An inter-component link is
the link of a variable to the variables of a different scope. An intra-component link is the link of a variable to the variables of the same scope. This web provides assistance in the adaptation of the existing software to make the concept of reusability system spawn, change, and evolve, rather than remain as a static library.

The **Alternative Web** consists of links of blocks in accordance with the functionality of each block. Only same phase documents, each of which are in different systems (bridge alternatives) or independent blocks (orphan alternatives), can be linked. This web provides a software developer with a choice of methods by which the same functionality is achieved. The information tag provides a software developer with information about each block in the Alternative Web.

### 3.4.7 Creation of Reuse Support Information

In order to show reuse support information created between two consecutive phases, we introduce a mapping function \( \mathcal{B} \) as follows:

\[
\mathcal{B} : P \times Q \rightarrow \{\text{RSI}\},
\]

where

- \( P \) is a phase in the software development life cycle,
- \( Q \) is the next phase of \( P \) in the software development life cycle, and
(RSI) is a set of reuse support information produced in between phases $P$ and $Q$.

The RSIs created by the mapping function $\mathcal{B}$ parameterized by two adjacent phases are as follows:

\begin{align*}
\mathcal{B}(U, R) &= \text{Horizontal Web}, \\
\mathcal{B}(R, D) &= \text{Horizontal Web}, \\
\mathcal{B}(D, P) &= \text{Horizontal Web}, \\
\mathcal{B}(P, C) &= \text{Horizontal Web}, \text{ and} \\
\mathcal{B}(C, T) &= \text{Horizontal Web}.
\end{align*}

The mapping function $\mathcal{J}$ shows a phase of software development life cycle and reuse support information that must be generated within a phase. The mapping function $\mathcal{J}$ is defined as follows:

\[
\mathcal{J} : P \rightarrow (\text{RSI}),
\]

where

- $P$ is a phase of the software development life cycle, and
- (RSI) is a set of reuse support information produced in a phase $P$.

The RSIs this mapping function yields for each phase is as follows:
\( J(U) \) = Semantic Web, Vertical Web, and Alternative Web in the user specification,

\( J(R) \) = Semantic Web, Vertical Web, and Alternative Web in the requirement specification,

\( J(D) \) = Semantic Web, Vertical Web, and Alternative Web in the design,

\( J(P) \) = Semantic Web, Vertical Web, and Alternative Web in the pseudo-code, and

\( J(C) \) = Semantic Web, Vertical Web, Syntactic Web, and Alternative Web in the code.

Mapping functions \( B \) and \( J \) are conceptually illustrated in Figure 3.15. In each phase, a Semantic Web, a Vertical Web, and an Alternative Web are created as reuse support information within each phase. In the coding phase, the only phase in which syntax is involved, a Syntactic Web is additionally created. On the other hand, between two adjacent phases, a Horizontal Web is created to support software reuse.
Figure 3.15  Reuse Support Information
4. IMPLEMENTATION OF THE QUINTET WEB

4.1 Environment and Characteristics

The Quintet Web has been implemented using C++. The object-oriented method has been applied because the characteristics of the Quintet Web are basically hierarchical.

Inheritance, which is an essential feature of the object-oriented method, is also essential to the Quintet Web. In the Quintet Web, documents in every phase are broken down into hierarchical blocks. The attributes of a parent block are inherited by its child blocks unless its children are characterized by other attributes. For instance, when a block of software is written in Pascal, its child blocks are assumed to be written in Pascal unless specified otherwise.

Encapsulation is also significant because the Quintet Web allows users to traverse any child web, i.e. the Semantic Web, the Horizontal Web, the Vertical Web, the Syntactic Web, and the Alternative Web, at any time and at any place in the Web. Consequently, the information in each Web may be accessed by other Webs, but must not be eventually updated.

Overloading may be applied to the Quintet Web. For example, methods to browse a block of a document in
different phases may be different. In order to browse
documents of six phases, six different functions may have
to be defined. When the object-oriented method is
applied, all six functions may be defined to have same
name with different types of parameters. When a function
is called, one of the six functions is called according to
the type of its parameter.

In addition to overloading, polymorphism may also be
applied. If the six functions are defined as the virtual
functions with the same formal parameter list but
different return types in different object, the function
call is made according to its base class. However, both
polymorphism as well as overloading are implementation
dependent. If all phases have the same file structure,
the application of overloading or polymorphism feature is
not necessarily required.

In this chapter, we discuss the implementation of the
Quintet Web. First, the overall strategy of the
implementation and the design method is presented. A
discussion of the implementation of each Web then follows.

4.2 Reuse Support Information in the Quintet Web

In order to implement the Quintet Web, two types of
information are required in addition to the reusable
software itself: global information and local information.
The global information includes information about reusable
software stored in the library, keywords that describe the software blocks, and attributes that characterize the software. Local information is local to each software component. Figure 4.1 illustrates reusable software and the two types of information. Global information includes attributes of reusable software, keyword lists, and general information of software to be reused. Reuse support information is the information analyzed as each Web in the Quintet Web. Reusable software is software of all phases that have been previously analyzed and stored in the software library.

Figure 4.1 Information for the Quintet Web
In order to implement the Semantic Web, links from keywords to blocks of software are required. For the Horizontal Web, links from blocks of software to their other phase documents are needed. To implement the Vertical Web, links among hierarchically structured blocks are required. The Syntactic Web requires the links of variables from their declaration to their uses. The Alternative Web needs links from blocks of software to their alternative blocks. In addition, the software that is to be reused must be stored in the database.

We implemented both types of information as well as reusable software as files. In the next sections, we discuss the information files and the software database.

4.2.1 Information File Structures

In order to store three types of global information, we created three files: a keyword file, a software list file, and a software attribute file. The keyword file includes keywords and their identification numbers (ids). The software list file includes keyword ids and ids of the software that are described by the keyword. The software attribute file includes software ids and attributes of each software.

The keyword file is used to identify the id of a keyword entered by a user. There are two ways a user can input a keyword. One way is to browse a keyword list and
select a number that corresponds to the keyword. The other way is to type a keyword. By selecting a keyword by either method, the keyword is identified as a keyword id. Instead of the keyword, the keyword id is used throughout the implementation.

Keywords in the keyword file are ordered alphabetically so that when they are displayed on the screen, a user can easily locate the keyword for search. In our implementation, a keyword consists of a combination of words. Each keyword has a unique id. In our implementation, a keyword consists of a combination of words. Each keyword has a unique id. When a new keyword is added to the file, a new id is automatically assigned to the keyword, and the keyword is inserted in the file. Although the sequence of keywords may be changed after the insertion, a unique id always designates its keyword. The keyword file is of the form:

\[
\left\{(K_0 \ Keyword_0),(K_1 \ Keyword_1), \ldots,(K_n \ Keyword_n)\right\},
\]

where

- \(K_i\), \(0 \leq i \leq n\), is a keyword id represented by an integer, and
- \(Keyword_i\), \(0 \leq i \leq n\), is a keyword.

The keyword file \(F\) is formally defined below.

\[
F = \left(\bigcup_{i=0}^{n} (K_i \ Keyword_i)\right) \land \\
\left(\forall_{0 \leq j,k \leq n} [ (j < k) \rightarrow (Keyword_j < Keyword_k) ] \right)
\]
The following type definitions are used to generate a keyword and its id pair.

```c
const int KeywordLength = 100;
typedef char KeywordType[KeywordLength];
typedef struct {
    int KeywordId;
    KeywordType Keyword;
} KeywordRecord;
```

The keyword file is created by writing this record to the binary file.

The software list file consists of keyword ids and their corresponding software ids. Like keyword ids, software ids are implemented as integers. Every software component in the database of the Quintet Web is assigned a unique software id. Using the keyword id that is identified by the user's selection, all software in the database can be easily located through this file. Note that, in the software list file, both keyword ids and software ids are implemented as integers. There must be a way to differentiate a keyword id from a software id, because a keyword id is followed by at least one software id in order to list the software in the database that are described by the keyword. In order to distinguish a keyword id from a software id, a leading special code control-K ('^K') is used to designate a keyword id. Any id following ^K is a keyword id. The software list file is of the form:
[^KK_0 \ S_0, \ldots, S_n \ ^KK'_0, \ S_0, \ldots, S_n, \ldots \ ^KK''_0, \ S_0, \ldots, S_n],

where \(^K\) is a control-K,
\(K_0, \ K_0', \ K_0''\) are keyword ids represented by integers, and \(S_0, \ S_n, \ S_0', \ S_n', \ S_0'', \ S_n''\) are software ids represented by integers.

The software attributes file consists of software ids and attributes of the software. When software is examined as a candidate to be reused, the software attributes are displayed on the screen so that a user can check if the software fits the requirements of the new software. A software id in the software attributes file is distinguished from the list of attributes by a leading special code control-S (\(^S\)). Software attributes are represented by free format texts. The software attributes file is of the form:

[^SS_0 \ A_0, \ldots, A_n \ ^SS'_0, \ A_0, \ldots, A_n, \ldots \ ^SS''_0, \ A_0, \ldots, A_n],

where \(^S\) is a control-S,
\(S_0, \ S_n, \ S_0', \ S_n'\) are software ids represented by integers, and \(A_0, \ A_n, \ A_0', \ A_n', \ A_0'', \ A_n''\) are software attributes.

The three types of information files discussed above include information the Quintet Web uses in order to retrieve reusable software. These files do not include the actual software to be reused.
4.2.2 Reusable Software Database Files

The reusable software database consists of separate files. Each software entity consists of multiple files so that documents of different phases are stored separately.

Each file is broken down into at least one block. Each block is embedded by a pair of block ids that are represented by integers. In order to differentiate the beginning and the end of a block, two different types of ids are introduced. A beginning block id is a block id leaded by a special character control-B (^B) and an ending block id is a block id leaded by another special code control-E (^E). Each phase document of a software entity is of the form:

\[
[^{BB}_0 text^{EB}_0], \text{ where}
\]

^B is a special code control-B,
^E is a special code control-E,
B_0 is a block id, and
text is the contents of the file.

When a file consists of multiple blocks, the file is of the form:

\[
[^{BB}_0 text_0^{EB}_0^{BB}_1 text_1^{EB}_1 \cdots^{BB}_n text_n^{EB}_n], \text{ where}
\]

^B is a special code control-B,
^E is a special code control-E,
B_0, B_1, B_n are block ids, and
text_0, text_1, text_n are contents of corresponding blocks.

When a block is embedded in a larger block, the blocks are of the form:
[
\text{[^BB}_i^\text{text}_i, \text{^BB}_j^\text{text}_j^\text{^EB}_j^\text{text}_j^2\text{^EB}_j], \text{where} \n\text{^B and ^E are special codes,} \n\text{^B}_i \text{ and ^B}_j \text{ are block ids,} \n\text{^text}_i, \text{^text}_j, \text{ and ^text}_j^2 \text{ are the content of the} \n\text{block}_i, \text{ and} \n\text{^text}_j \text{ is the content of the block}_j. \n\text{^text}_i, \text{^text}_j, \text{^text}_j^2] \text{ is block}_i \text{ decomposed into three} \n\text{parts. ^text}_j \text{ is the contents of the block}_j, \text{ but it is} \n\text{also a part of the block}_i. \text{ This means, a part of the} \n\text{block}_i \text{ is assigned as a child block}_j. \text{ A block}_j \text{ is embedded} \n\text{in a block}_i. \n\text{Between a pair of block ids, any number of blocks can} \n\text{be embedded. This means, block}_j \text{ is decomposed into} \n\text{multiple blocks, i.e., block}_j^1, \text{ block}_j^2, \ldots, \text{ block}_j^m. \text{ So,} \n\text{^text}_j \text{ is broken down to multiple blocks ^text}_j^1, \text{^text}_j^2, \ldots, \text{^text}_j^m. \text{ The block}_i \text{ is of the form:} \n\text{[^^BB}_i^\text{text}_i, \n\text{[^^BB}_j^\text{text}_j^1^\text{^EB}_j^\text{^BB}_j^2^\text{text}_j^2^\text{^EB}_j^2 \ldots, \text{^text}_j^m, \text{^text}_j^m^\text{^EB}_j],} \n\text{Alternatively, a block may be embedded in a block} \n\text{which is already embedded in a larger block. In this} \n\text{case, a part of block}_j \text{ is assigned as a child block of} \n\text{block}_j. \text{ So, ^text}_j \text{ is broken down into three parts: ^text}_j^1, \text{^text}_k \text{^text}_j^2. \text{ The block}_i \text{ is of the form:} \n\text{[^^BB}_i^\text{text}_i, \n\text{[^^BB}_k^\text{^text}_k^\text{^EB}_k^\text{^text}_j^2^\text{^EB}_j \text{^text}_j^m^\text{^EB}_j],} \n\text{Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.}
This scheme is same as the nesting scheme of pairs of parentheses except in one instance. A set of properly nested parentheses is a context-free language. Therefore, among a sequence of open and close parentheses, there is only one way to match an open parenthesis to its closing parenthesis. However, in the blocking system of the Quintet Web, every opening block id and end block id are identified by their block ids. Therefore, blocking in the Quintet Web allows the overlap of texts between two blocks. The following example will illustrate the difference. We have a sentence in the requirement specification that is to be broken down. Here is the text:

Open file ••• in writing mode ••• check if the file is open successfully•••.

We want to break down this sentence into two parts: one is to state open file in writing mode and the other is to state check if the file is successfully opened. However, the second block must be precise enough to state to check the file is opened in writing mode. In the text, "in writing mode" must be included in both blocks. If parentheses are used, blocking can be done as follows:

(Open file ••• (in writing mode) ••• check if the file is open successfully•••).
The first "(" has to match the first ")" and the second "(" has to match the second ")". However, the first "(" can not match the first ")“. Instead, the first "(" matches the second ")". So, the text "in writing mode" can not be included in two consecutive blocks. The block id matching in the Quintet Web differentiates all beginning block ids and all end block ids. So, the sentence can be decomposed as following:

^B1 Open file… ^B2 in writing mode ^E1 … check if the file is open successfully… ^E2.

This decomposition can separate two blocks as follows:

[^B1 Open file … in writing mode ^E1], and
[^B2 in writing mode … check if the file is open successfully… ^E2].

Now, the second block includes the information that open file operation is done in writing mode.

In the following sections, we discuss the design and the implementation of the Quintet Web and its five sub-webs.

4.3 Design of the Quintet Web

The Quintet Web was implemented using the object-oriented paradigm. The highest class is the Quintet Web consisting of two member data and four member functions. Declaration of the Quintet Web is shown below.
class QuintetWeb {
private:
    // member data
    int SoftwareId;     // software id
    int KeywordId;      // keyword id
public:
    // constructor
    QuintetWeb ();
    // member functions
    int GetSoftwareId() {return SoftwareId;}  
    void SetSoftwareId (int SWId) {SoftwareId = SWId;}
    int GetKeywordId() {return KeywordId;}    
    void SetKeywordId (int KeyId) {KeywordId = KeyId;}
}

SoftwareId identifies which software is being currently examined. KeywordId identifies the keyword selected by the user to examine the software that are described by the keyword. Member function GetSoftwareId returns SoftwareId. SetSoftwareId initializes or updates SoftwareId. GetKeywordId returns KeywordId. SetKeywordId initializes or updates KeywordId. Only these four member functions can access SoftwareId and KeywordId because data members are declared as private.
Overall design of the Quintet Web is illustrated in Figure 4.2. The two member data declared in the Quintet Web are inherited to its sub-classes, i.e., the Semantic Web, the Horizontal Web, and the Syntactic Web, via member functions. The Semantic Web and the Horizontal Web are implemented in one sub-class of the Quintet Web. Member data declared in the Semantic/Horizontal Web sub-class are inherited to the Vertical Web and to the Alternative Web. The sub-classes are discussed in the following sections.
4.4 Design of the Semantic Web and the Horizontal Web

The Semantic Web and the Horizontal Web are implemented as a sub-class of the Quintet Web. Therefore, both SoftwareId and KeywordId are inherited from the Quintet Web. The Semantic Web identifies software that is described by a keyword that a user has selected. Once a software is identified, control is transferred to the Horizontal Web. The Horizontal Web displays a block of the document. The default phase is the user specification, but the user can specify any phase to display in order to examine the software. If the user wants to examine the information in any of the other webs, control can be transferred to that web. If the user wants to examine the other software available, control is transferred back to the Semantic Web. If the user wishes to examine the vertical structure of the software, the Vertical Web is invoked. If the user wishes to examine the alternative software to the software currently being examined, the Alternative Web is invoked. The declarations of the Semantic/Horizontal Web are shown below.

```c
const int PhaseNumber = 6;
typedef int BlockRecordType[PhaseNumber];
enum Phase {UserSpec, ReqSpec, Design,
            Pseudocode, Code, TestData};

// Semantic Web and Horizontal Web inherit from QuintetWeb
class SemHorWeb : public QuintetWeb {
private:
    // member data
    BlockRecordType BlockRecord; // record of block ids
    int BlockId;                 // block id
    Phase PhaseId;               // phase id

public:
    // constructor
    SemHorWeb();
    // member functions
    void GetBlockRecord (BlockRecordType *);
    void SetBlockRecord (BlockRecordType *);
    int GetBlockId () {return BlockId;}
    void SetBlockId (int BlkId) {BlockId = BlkId;}
    Phase GetPhaseId () {return PhaseId;}
    void SetPhaseId (Phase PID) {PhaseId = PID;}

The Semantic/Horizontal Web classes maintain three data members: BlockRecord, BlockId, and PhaseId. Every block in a phase document is identified by the BlockId represented by an integer. BlockId is the id of the block currently being displayed. PhaseId identifies which phase document the block is. BlockRecord is an array of six block ids. BlockRecord keeps block ids of the other phase documents that will be displayed as the user requests. The Horizontal Web uses this record to display different phase documents according to the user's request.

4.5 Design of the Vertical Web

The Vertical Web inherits BlockRecord, BlockId, and PhaseId from the Semantic/Horizontal Web. SoftwareId and KeywordId are also inherited from the Quintet Web via the Semantic/Horizontal Web. The Vertical Web declaration is shown below.
const int VerticalNumber = 100;
typedef struct{
    int BlkId;
    int VerticalBlock[VerticalNumber];
} ListType;

// The Vertical Web publicly inherits from
// the Semantic/Horizontal Web, but
// ultimately, from the Quintet Web

class VerticalWeb : public SemHorWeb {
private:
    // member data
    ListType ParentList; // array of parent block ids
    ListType ChildrenList; // array of children block ids
public:
    // constructor
    VerticalWeb ();
    // member functions
    void GetParentList (ListType *);
    void SetParentList (ListType *);
    void GetChildrenList (ListType *);
    void SetChildrenList (ListType *);
}

The Vertical Web declares, defines and maintains ParentList and ChildrenList. ParentList is a list of the parent blocks of the block currently being examined. ChildrenList is a list of the children blocks. In our implementation, only the information about the current block is kept in the ParentList and ChildrenList. If the user wishes to examine other children or parent blocks, these two list would be referenced.

N-ary tree is a seemingly tempting data structure to implement the Vertical Web, but there are two reasons why the Vertical Web cannot be implemented by n-ary tree.
First and most importantly, while a child node in an n-ary tree can not have more than one parent, the Vertical Web allows more than one parent block for a child block. For example, in the object-oriented method, multiple inheritance is allowed to a sub-class object to inherit from more than one super-class objects. Secondly, while the n-ary tree operations implicitly assume the modification of the tree via insert, delete, and reorganize, the Vertical Web operations are not intended to modify the existing database of reusable software. The Vertical Web searches and displays an existing and unchangeable software blocks.

When the Vertical Web takes the control, SoftwareId, KeywordId, BlockId, and PhaseId are inherited. So, the Vertical Web locates current block's (identified by SoftwareId and BlockId) parent or child block of a phase (identified by PhaseId), and displays the block. As the user requests traversing up or down, the Vertical Web searches the corresponding block to display. If the user wishes to display other phase document, control is transferred back to the Horizontal Web. If the user wishes to locate different software described by different keyword, the control is transferred back to the Semantic Web.
4.6 Design of the Alternative Web

The Alternative Web inherits from the Semantic/Horizontal Web. SoftwareId and KeywordId are also inherited from the Quintet Web. The Alternative Web declares, defines, and uses AlternativeList. The definition of the Alternative Web is shown below.

\[
\text{const int AlternativeNumber = 100;}
\]
\[
\text{typedef struct}{
  \text{int BlkId;}
  \text{int AlternativeBlock[AlternativeNumber];}
}\}
\text{ListType;}
\]

// The Alternative Web publicly inherits from // the Semantic/Horizontal Web, but // ultimately, from the Quintet Web
class AlternativeWeb: public SemHorWeb {
  private:
    // member data
    ListType AlternativeList; // array of alternative // block ids
  public:
    // constructor
    AlternativeWeb();
    // member functions
    void GetAlternativeList (ListType *);
    void SetAlternativeList (ListType *);
}

The Alternative Web first sets alternative blocks information to AlternativeList, then, searches alternative blocks and displays one by one. Since the Alternative Web inherits PhaseId, SoftwareId, and KeywordId, it can search the right phase document blocks of the software being examined. Similarly, in the Vertical Web, when a user
wishes to examine other software, control is transferred
back to the Semantic and Horizontal Web.

4.7 Design of the Syntactic Web

The Syntactic Web inherits SoftwareId and Keyword
from the Quintet Web. The member data declared and
defined in the Syntactic Web are VariableId and
StatementList. Definition of the Syntactic Web is shown
below.

const int StatementNumber = 100;
typedef struct{
    int VarId;
    int StatementList[StatementNumber];
} ListType;

// The Syntactic Web publicly inherits
// publicly from the Quintet Web

class SyntacticWeb: public QuintetWeb {
    private:
        // member data
        ListType StatementList; // array of statement number
    public:
        // constructor
        SyntacticWeb();
        // member functions
        void GetStatementList (ListType *);
        void SetStatementList (ListType *);
}

When the Syntactic Web takes control, StatementList
is generated. The only phase that is involved in the
Syntactic Web is code because no syntactic relationship
exists in other phases.

When a user wishes to examine the syntactic
relationship in a block of code, a list of variable is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
displayed on the screen. The user can select by the number or input the variable. The Syntactic Web searches from the definition of the selected variable to all occurrences of the variable. The Syntactic Web displays blocks of code with marks left of statements that use the variable specified by the user.

4.8 Global and External Functions

The Quintet Web has been implemented employing global functions that are not written for a specific Web. These functions are defined externally. Below are list of the function prototypes.

```c
void DisplayDocument (int SystemNbr, int BlockNbr, Phase PhaseNbr);
Phase GetUserPhase ();
void GetFileName (Phase PhaseNbr, int SystemNbr);
void CreateFile (int SystemNbr, BlockRecordType *);
```

DisplayDocument displays a block of document using System id, Block id, and Phase id. GetUserPhase allows a user to select a phase to be displayed. GetFileName inputs file name to be created by CreateFile function. CreateFile copies all phases document block to new files. File names are input by the user. Otherwise, default file names are used.

4.9 Prototype of the Quintet Web

In this section, we demonstrate the execution of the prototype of the Quintet Web introducing the sequence of
screens. In implementation, the five webs in the Quintet Web are hidden to users. In other words, users are not required to have knowledge about the internal implementations of the Quintet Web. Each screen is shown in simplified form.

Enter a Choice: _
1: Browse Keywords
2: Enter a Keyword

Figure 4.3 Initial Screen of the Quintet Web Prototype

The initial screen is shown in Figure 4.3. If a user chooses 1, a keyword list is given to the user. A sample keyword list is shown in Figure 4.4.

1: ASCII File - Create
2: ASCII File - Open
3: ASCII File - Read
4: ASCII File - Write

Enter a Keyword Number: _

Figure 4.4 Keyword List

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Attribute of Software

This is a complete program.
Reads/Writes a text file by character.
Programming Language: Borland C++ 3.1
Running System: IBM or compatible PCs
Processor Type: 80286 or higher
Program Size: 32 lines
Time Complexity: O(n)
Space Complexity: O(1)

Examine This System? (y/n)

Figure 4.5 Attribute Tag

By entering a keyword number, a software component is selected and its attribute tag is shown. An example attribute tag is shown in Figure 4.5. If the attributes are appropriate for the software to be developed, a user answers yes('y') to the prompt. Then, the choice of the phase to be browsed is prompted. This screen is shown in Figure 4.6.
Enter a Phase to be Browsed:

1: User Specification
2: Requirement Specification
3: Design
4: Pseudo-code
5: Code
6: Test Data

Figure 4.6 Choice of Phases

Enter a Choice:

1: Create Files.
2: Browse Another Phase Document.
3: Browse Alternative Software.
4: Browse the Parent Block.
5: Browse the Child Block.
6: Browse the Chain of a Variable.
7: Examine Other Keywords.
8: Finish the Quintet Web.

Figure 4.7 Choice of Webs

So far, a user has selected a keyword, and examined the attributes of a software component through the Semantic Web. By selecting a phase from the screen shown in Figure 4.6, the Horizontal Web is invoked. By each choice, a block of the appropriate document is shown on the screen. The actual screen is omitted because it is simply a copy.
of the actual document. The next screen, shown in Figure 8, guides the user in the selection of the web to choose for browsing.

By selecting 1, a copy of a software block is created for all phases as text files. When these files are created, special codes are eliminated so that they can be used as they are. A user can specify file names; otherwise, default file names are used. The screen to input the file name is not shown here because it is a simple list of default file names for each phase and prompt to the user alternate file names.

Choice 2 invokes the Horizontal Web again. The screen shown in Figure 4.6 will appear and same process explained above repeats.

Choice 3 invokes the Alternative Web. The alternative software is browsed on the screen. In this process, the phase a user selected on the screen shown in Figure 4.6 is used as the default phase. The screen shown in Figure 4.7 appears again.

Choices 4 and 5 invoke the Vertical Web. Callers/callees, or superclasses/subclasses of current software block are browsed. The phase a user selected in Figure 4.6 is used as the default phase. If no callers/callees or superclasses/subclasses are available,
appropriate messages are issued and goes back to screen in Figure 4.7.

Choice 6 invokes the Syntactic Web. Users are prompted to enter or select a variable name. This screen is shown in Figure 4.8. Figure 4.9 is the screen for a variable name list. By either method, a variable is selected. The chain of a variable $fp$ is shown in Figure 4.10. In our implementation, each statement where the selected variable occurs is marked in order to show the location of the variable.

```
Enter a Choice: _

   1: Browse Variable Names
   2: Enter a Variable Name
```

Figure 4.8 Options for Selecting a Variable Name

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
main(int argc, char *argv[]) 
{
    FILE *fp;
    char buffer[256];
    ...  
    fp = fopen(argv[1], "r"); 
    if (!fp) 
    { 
        ...  
    }
    while (fgets(buffer, 255, fp) != NULL) 
    { 
        cout << buffer;
    }
    fclose(fp);
    return 0;
}

Figure 4.9 Variable List

Figure 4.10 Chain of Variable fp
Returning to the initial screen in Figure 4, choice 7 is to be selected. A search for other software components starts again.

Choice 8 ends the Quintet Web.
5. SUMMARY AND CONCLUSIONS

The concept of software engineering and software reuse were born together. In 1968, McIlroy suggested the establishment of a "components factory" that can provide "mass produced software components [MCIL 1969]" to achieve software that can be engineered reusing the components. For software to be engineered, software engineers must have parts. The parts (or components) are nothing but reusable software artifacts, analogous to "families of structural shapes, screws, or resistors [MCIL 1969]" for engineers. Initiated by his concept of "components factory," software reusability has become recognized as an important method [BOLD 1990b] to enhance software productivity and quality, and, ultimately, to overcome the ongoing software crisis. Many successful practices are reported, [INCO 1990] [ISOD 1992] [PRIE 1993] but the impact is not yet so significant as promised. There remains considerable unexploited software reuse potential. [BANK 1993]

As the study of software reusability matures, the definition of software reuse has also changed. Reuse of source code level [MCIL 1969] has extended to the reapplication of a variety of kinds of knowledge [BIGG 1987]. Pessimism on the reuse of code was due to the insignificance of the impact gained by reusing source code.
because the estimated cost for the coding phase to the overall effort has been reported as only 13%. [HORO 1989]

Simply, software reuse is defined as the activity of reusing existing software for a new purpose. [INCO 1990] The concept of existing software has also expanded to more precise definitions. Traditionally, it means the software currently exists. W. Tracz addressed that software must be designed to be reused. [TRAC 1990] R. Banker et al. also emphasized that software development methodology has to focus on the reuse. [BANK 1993] That is, software being developed (or engineered) currently must aim for future reuse. Nowadays, the concept of existing software must include software that will exist in the future (because it is being developed currently) as well as software written in the past (therefore, it exists currently).

5.1 Summary of the Features of the Quintet Web

The literature on software reusability identifies the need for better reusability models, describes the characteristics that such models should possess, and supports the finding that such a model does not currently exist. The characteristics that were identified that a reusability model should possess are:

1) Software must be designed to be reused,
2) A software reuse activity must require neither an intensive initial investment nor excessive ongoing effort,

3) A software reusability system must support all phases of software development process,

4) A software reusability system must be independent from the software development methodology,

5) A software reusability system must easily be applied to the standards of each organization,

6) A software reusability system must support software maintenance as well as software development, and

7) A software reusability system must support adaptation of existing software to a new environment.

A description of how the Quintet Web satisfies each of the seven criteria is given below.

1) Software must be designed to be reused. The Quintet Web can be, and ought to be applied to the software development as well as software reuse process because software reuse effort must start with the software development process. In other words, the software development process must be the process of developing reusable software. In the report of software reuse case
studies, Incorvaia and Davis [INCO 1990] discussed this aspect as follows:

In the more successful programs, reuse was more than just a concept espoused by individuals. Reusing software was a way of life. This is especially true in organizations, ..., where most developers do not even think of what they were doing as software reuse. To them, it is simply how to develop software.

Through our implementation of the Quintet Web, we realized that the generation of the Reuse Support Information (RSI) was by no means trivial. We had to analyze the previously developed software to reconstruct the evolutorial relationships between phases as well as inter-/intra-structural relationship of the software. It was a time consuming and an error prone process as Tracz pointed out. [TRAC 1988b]

The concurrent generation of the RSI along with the software development process will greatly reduce the effort of software reuse activity because analysis of inter-/intra-relationship of the software is also a part of the development process. It is also clear that the automation of the RSI generation process will relieve the burden and enhance the precision of the RSI. If the RSI is generated when the software is being developed, no sophisticated technique is required. Simply, documents are decomposed
into blocks according to operational functionalities (being a part of development process), then logical links are made.

2) A software reuse activity must require neither an intensive initial investment nor excessive ongoing effort. One of the inhibitors of software reuse is that software reusability is believed to be a commitment for a long term investment and long term return on investment. [WEGN 1983] [HOOP 1991] However, it may not be a capital-intensive software technology. With the application of the Quintet Web to the development environment, software reuse activity starts with development. There is no need to establish a components factory that contains huge amount software parts from the beginning. The database will grow as software development process goes on. At the beginning of reuse activity, the reuse ratio may be low due to the small number of collection in the database. However, regarding the magnitude of software being developed every year, the growth of the collection in the database is not always pessimistic.

If the Quintet Web is applied to the development process, the Quintet Web will encourage software developers to follow the predefined standard in their organizations. The idea of software reusability will always be explicit, and become a way of life. Incorvaia and Davis confirmed this aspect in their case studies report [INCO 1990]:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Reusable components must be built with reuse in mind; reusability cannot be retrofitted later.

As a side effect of the RSI generation process, but very important, the consistency of the software being developed can be checked if software developers develop software and generate RSI in a synchronized manner applying the Quintet Web, because the Horizontal Web requires every block in a document to be linked to its adjacent phases according to their functionalities.

3) A software reusability system must support all phases of software development process. Many researchers address that documents of all phases must be reused in order to maximize the benefit of software reuse. [BANK 1993] [ISOD 1992] [BIGG 1989] The Quintet Web enables software developers to reuse software of all phases of the development life cycle. At the same time, while software developers perform a keyword search, this feature encourages the examination of various phase documents. Understandability of software is enhanced through the investigation of software documents at different level of abstraction. Especially, reuse at an early stage of development life cycle will benefit.

4) A software reusability system must be independent from the software development methodology. As Incorvaia and Davis pointed, establishment and use of standards of
software are considered to be essential. [INCO 1990] This aspect has been confirmed by the case studies conducted by them. If a standardization of the software development process is a necessity, the standardization must be essential for software reusability. However, in the real world of software, different software development methodologies are employed by different organizations. There is no unique methodology that is utilized as a standard for all environments. Moreover, a standard may change over time. Software tools, not limited to reuse tools, require standardization. At the same time, it is necessary to be flexible in order to move from one environment to another. However, this requirement may not be considered as a contradiction. We can achieve both aspects through the implementation of the Quintet Web.

In our implementation, we used a waterfall model that consists of six phases: user specification, requirement specification, design, pseudo-code, code, and testing. From this software development life cycle, deletion or addition of a phase was achieved easily. Insertion of a phase was involved in an additional RSI generation, but it did not require new or advanced technology. A division of a phase into multiple phases, for instance, division of the design phase into functional and detailed design phases, was done as the insertion of a phase. These are examples of some
irregularities imposed to the Quintet Web, but we confirmed that the Quintet Web can be adapted to a new environment.

5) A software reusability system must easily be applied to the standards of each organization. The Quintet Web encourages standardization in two ways. Once the Quintet Web is set up for a development process, it will try to link a block of document to other blocks of the same phase or adjacent phases. If the link cannot be established, this will be considered as an exception. About exceptions, software developers must provide reasons that will explain and justify the exceptions to the users of the reusability system. Alternatively, if the exception is illegitimate, i.e., is not intended, a request to update that portion of software is given. Secondly, if the way the Quintet Web is set up is not suitable to a development process, the set up must be modified so as to be justifiable. This discourages an individual developer or a project group to alter the organizational standard arbitrarily. While organizational structure is not an important factor for software reuse, well-defined software engineering methods and standards must be in place. [INCO 1990]

The search, in order to locate the appropriate reusable software, is considered as a vitally important factor for successful software reuse. [BANK 1993] The Quintet Web

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
allows a keyword search. However, a combination of keywords may not be enough to describe software. Moreover, the requirements for software often cannot be easily translated into a combination of keywords. A software developer seeking reusable software usually calls upon many forms of knowledge, such as knowledge of the domain or subdomain. [DUNN 1993]

The Quintet Web provides its user the attributes of each reusable software candidate prior to in-depth examination to the software. Attributes of the software may be included in the search in order to extend the keyword search capability. Although we implemented the attributes tag with text files that consist of natural language descriptions, it can take different forms, for instance, keywords, or combination of keywords and natural language expressions. When the attributes do not fit to the software developers' needs, alternative software blocks may be provided. Since a variety of algorithms can be employed to accomplish an identical operation, the capability to provide the alternative software gives software developers higher possibility to locate an exact match.

6) A software reusability system must support software maintenance as well as software development. Design evolution and maintenance is considered as the dominant activity in the software development process. The estimated
cost of design evolution in industry accounts for 70% to 90% of a software system. [ARAN 1993] Knowledge of the hierarchical structure of software is important to understand the design of the software. The Vertical Web provides software developers a means to examine a hierarchical structure. Through the Vertical Web, a software developer is provided the ability to decide the extent of reuse. After examining a block of software, its higher or lower hierarchy can also be examined. If the higher hierarchy suits his needs, a software developer may decide to reuse larger extent of software.

Regarding the effort made for maintenance being 60% to 75% of software cost [HORO 1989], software reuse activity must also be applied to software maintenance. The Quintet Web can be applied to the maintenance process. For example, for corrective maintenance, the block to be updated can be replaced by the block in the database. A set of blocks can be added with other phase documents for perfective maintenance.

7) A software reusability system must support adaptation of existing software to a new environment. Software environments vary. Therefore, in many cases, software artifacts must be modified and/or adapted to the new environment after they are retrieved. The Syntactic Web supports adaptation by providing syntactic structure of a
software. When two blocks of code are composed into one, conflicts of variables may cause a complication in reuse. The syntactic structure of each block of code helps software developers to differentiate the variables in conflicts. The Alternative Web provides choices of software to meet different requirements of the new system.

The Quintet Web model successfully addresses each of these seven characteristics. Through the implementation of the model, we have demonstrated that the model in fact works as proposed.

5.2 Contributions of the Research

The contributions and the collectively unique features of the Quintet Web are summarized as follows:

1) Consistency and validity of software can be checked throughout the phases of software development process when software is being developed,

2) Software can be designed to be reused without affecting any software development phase,

3) Both software development and maintenance are supported,

4) The Quintet Web supports all stages, from the early stages to the latest, of the software development process,
5) A broad range of software artifacts can be reused, i.e., specifications, design, pseudo-code, test data, and code,

6) The extendibility of reuse from a small portion of software to a larger portion is possible,

7) Five relationships dispersed in software are condensed into RSIs and the RSIs are interwoven, to structure a centralized software library in order to support the integration of different software concepts,

8) Software developers have choices of different algorithms for the identical operation,

9) The model is independent from software development methodologies,

10) The Quintet Web supports adaptation of the software to be reused,

11) Users of the Quintet Web are not required to have knowledge about each Web,

12) Application of the model does not require a large amount of initial investment,

13) Software developers (or organizations) are encouraged to standardize software activities,

14) The model promotes software reuse as the standard method to develop software, and
15) The web can be easily applied to new software being, or to be, developed. The Quintet Web can also be applied to the existing software.

5.3 Future Research

During the implementation of the Quintet Web, we gained insights into areas to study in order to enhance the practicality of the application. First, the search methodology should be expanded. The Quintet Web is initiated by a keyword search to locate a block of software. Therefore, the search process is of great importance.

Secondly, an application of non-linear database structure, for instance, hypertext, is possible. Graphics application could also be supported. The Quintet Web enables software developers to examine documents of different phases at any time. Specifications and design documents tend to utilize more and more graphics application. Application of hypertext can extend the browse process with a variety of data types including text and graphics data.

Third, interface with other software must be addressed. Virtually all types of software can be documented into a variety of machine readable forms. In order to control different documents created by different tools, the interface of the Quintet Web with other systems output need to be studied.
Fourth, the data structure of the centralized database (component library) must be exploited in depth. This will enhance the efficiency and the effectiveness of the Quintet Web system.

Fifth, in order to enhance the adaptation and the synthesis process of the software to be reused, text and graphics editing feature could be included the model.

Finally, a methodology to engraft the Quintet Web model with other reuse technologies will expand the capability. Transformation systems and code generation systems are examples of such systems. When a software developer finds a block which is written in Pascal, he may want to translate the block into C. Interface with a transformation system will resolve the software developer's problem. Graft with code generation systems will be essential when a retrieved block is written in a specification language and it must be translated into a programming language.
REFERENCES


also in Proceedings of COMPCON S'87, 1987, pp.376-379


also in

137

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Proceedings of the Workshop on Software Reusability and Maintainability, The National Institute of Software Quality and Productivity, October 1987


also in Proceedings of the Workshop on Software Reusability and Maintainability, The National Institute of Software Quality and Productivity, October 1987


and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


[BOWL 1983] abstract only


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


also in


also in


also in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


Engineering, Association of Computing Machinery, 1987, pp.369-376


also in Proceedings of the Tenth Minnowbrook Workshop on Software Reuse, 1987


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

also in Proceedings of the Workshop on Software Reusability and Maintainability, The National Institute of Software Quality and Productivity, October 1987


also in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Proceedings of the Workshop on Software Reusability and Maintainability, The National Institute of Software Quality and Productivity, October 1987


also in
and


also in
and
Tutorial: Software Reusability, Freeman, Peter, ed., The Computer Society of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


also in
and


also in
Proceedings of the Workshop on Software Reusability and Maintainability, The National Institute of Software Quality and Productivity, October 1987


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


also in Proceedings of the Workshop on Software Reusability and Maintainability, The National Institute of Software Quality and Productivity, October 1987


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
also in Communications of the ACM, April 1992, Vol. 35, No. 4, pp.55-64


also in Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences, 1987, pp.521-529


also in
Proceedings of the Workshop on Software Reusability and Maintainability, The National Institute of Software Quality and Productivity, October 1987


also in

also in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
also in GTE Journal of Services and Technology, Vol.1, Spring 1987, pp.23-31


Engineering Notes, Vol.16, No.4, October 1991, pp.60-63


also in Proceedings of the Workshop on Software Reusability and Maintainability, The National Institute of Software Quality and Productivity, October 1987


also in
Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences, 1987, pp.513-520


also in Proceedings of COMPCON S'87, 1987, IEEE, pp.358-363

also in ACM SIGSOFT Software Engineering Notes, Vol.13, No.1, January 1988, pp.17-21

also in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


VITA

Youngsuck Cho was born in Seoul, Korea, on May 13, 1954. He received a B.A. degree in Philosophy in 1978 from Sogang University, Seoul, Korea. From 1980 to 1984, he worked as a Programmer and as a Systems Analyst at FACOM Korea, Ltd. (presently, Fujitsu Korea, Ltd.), in Seoul. He received his Master's degree in Library and Information Science in 1988 from Louisiana State University. Since December 1988, he has been a graduate assistant and a Computer Systems Analyst in the Graduate School at Louisiana State University.

His research interests include software reusability, reverse engineering, reengineering, object-oriented paradigm, software development methodology, and information retrieval.
DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Youngsuck Cho

Major Field: Computer Science

Title of Dissertation: A Multi-Faceted Reusability Model: the Quintet Web

Approved:

[Signatures of Major Professor and Chairman, Dean of the Graduate School, and EXAMINING COMMITTEE members]

Date of Examination:

July 14, 1994