BRIDGE : A COMPONENT BASED SOFTWARE DEVELOPMENT LIFE CYCLE PROCESS MODEL TO CATER WITH THE PRESENT SOFTWARE CRISIS

A thesis submitted to the University of North Bengal
For the award of

Doctor of Philosophy

in
Computer Science and Application

BY
ARDHENDU MANDAL

SUPERVISOR
Prof. S. C. Pal

Department of Computer Science and Application
University of North Bengal

JULY, 2015

Dedicated to my parents

Late Abinash Ch. Mandal
and
Smt. Lilabati Mandal

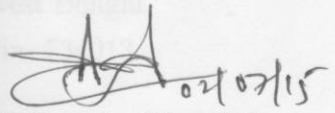
Acknowledgment

I am indebted to a large number of people for their inspiration and support during this research work and writing of my doctoral thesis.

I would like to express my deep sense of gratitude to my supervisor Dr. S. C. Pal, Professor, Dept. of Computer Science and Application, University of North Bengal for his valuable guidance and inspiration during the preparation of the thesis.

I would also like to express my sincere thanks to the Head, all the faculty members and staff of the Dept. of Computer Science and Application, University of North Bengal for providing facilities and cooperation.

My heartfelt thanks go to my mother Smt. Lilabati Mandal, my wife Smt. Nupur Mandal (Sarkar) and other family members for their love, care, understanding and for keeping faith on me to complete the research work. Above all, I would like to express my sincere respect and gratitude to my father, Late Abinash Ch. Mandal who guided and taught me over his lifetime with love and zeal. He will always remain deep within my heart as an eternal source of inspiration.



Ardhendu Mandal
Department of Computer Science and Application,
University of North Bengal,
Raja Rammohunpur,
P.O.- N.B.U.
Dist- Darjeeling,
West Bengal,
Pin- 734013

Declaration

I herewith declare that the thesis entitled **BRIDGE : A COMPONENT BASED SOFTWARE DEVELOPMENT LIFE CYCLE PROCESS MODEL TO CATER WITH THE PRESENT SOFTWARE CRISIS** has been prepared by me under the supervision of Dr. S. C. Pal, Professor, Department of Computer Science and Application, University of North Bengal. No part of this thesis has formed the basis for the award of any degree or fellowship previously.

02/07/15

Ardhendu Mandal

Department of Computer Science and Application,
University of North Bengal,
Raja Rammohunpur,
P.O.- N.B.U.
Dist- Darjeeling,
West Bengal,
Pin- 734013

DEPARTMENT OF COMPUTER SCIENCE AND APPLICATION
UNIVERSITY OF NORTH BENGAL

Dr. S. C. Pal
Professor

P.O. North Bengal University
PIN - 734013, India.
Phone (O) +91-353-2776344
Fax : +91-353-2699 001
E-mail : schpal@rediffmail.com

Ref. No. **Date :** 02/07/2015

CERTIFICATE

I certify that Sri. Ardhendu Mandal has prepared the thesis entitled **BRIDGE: A COMPONENT BASED SOFTWARE DEVELOPMENT LIFE CYCLE PROCESS MODEL TO CATER WITH THE PRESENT SOFTWARE CRISIS**, for the award of Ph.D. degree of the University of North Bengal, under my guidance. He has carried out the work at the Department of Computer Science and Application, University of North Bengal.

Schpal
02/07/15

Professor
Department of Computer
Science & Application
University of North Bengal

Dr. S. C. Pal
Professor

Department of Computer Science and Application,
University of North Bengal,
Raja Rammohunpur,
P.O.- N.B.U.
Dist- Darjeeling, West Bengal,
Pin- 734013

Abstract

The rapid development in hardware (HW) technology had made the modern computers to achieve higher computational speed with huge memory and storage capacity. Hence, the expected services from computer systems have reached to the level of almost to zenith. To utilize the available computational resources, the paradigm of software systems developments has also been shifted from the traditional approaches significantly. The modern software development projects are becoming more challenging over the time from the complexity, quality and cost point of view. The software development time remains one of the most critical issues as the HW technology of the targeted machines changes even before the completion of the software projects undertaken for those systems. Further, quality of the software systems is in crisis. Developing quality software within time and budget still remains a great challenge for the software development organizations. The increasing size and complexity of these modern systems are the primary reason for this challenge. Most of the software development projects over runs by budget. Hence, software development approaches and methodologies must change in accordance to the HW technologies consistently.

There exist several Software Development Life Cycle Models (SDLC) in literature and used by the practitioners in developing software (SW) systems. But, all of these SDLC models have their own merits and limitations. Further, studies shows that rarely the SDLC models are either used or suitable for the modern software development projects. Hence, to accommodate the industrial needs and best practices, new software development approaches and life cycle models need to be developed or existing SDLC models need to enhance periodically. Current research in software development life cycle models (SDLC) is beginning to emphasize the use of different industrial practices in software development.

This thesis presents an approach for software development following a new software development life cycle model proposed by us – named BRIDGE. The BGRIDGE life cycle model proposes an engineering approach for development of good, efficient, quality software systems within time and budget. This software engineering approach is developed keeping focus on object-oriented methodologies, component based software development methodologies and, incremental and iterative development process modeling. It supports the entire life cycle of software systems from feasibility study to maintenance and

includes project management, software development and quality management activities.

Then, we have discussed how the Agile software development philosophy may be achieved through BRIDGE- a traditional software development life cycle model. We also discussed the emergence of component based software development approaches. Further, we have discussed the suitability of the BRIDGE life cycle model for the modern software projects over the other available life cycle models with its comparative analysis.

Finally, we have concluded and recommended this process model to be used in real software development to alleviate the present software crisis upto a significant level.

Preface

For optimal utilization of computational resources, sufficient care has been taken during the development of software systems. To maintain the consistence pace with the advancement in Hardware (HW) technologies, necessary advancements in the software development approach and methodology is solicited. But, it has been seen that there remains many scopes for further advancements in the field of software development life-cycle models and process. Current research in software development life cycle models emphasizes the use of different best industrial practices in software development. This thesis presents a new software development life cycle model- named BRIDGE projected by us to address the modern software crisis. The BGRIDGE life cycle model enforces an engineering approach for development of good, efficient, quality software systems within time and budget with the primary focus on modern technologies and methodologies i.e. object-oriented methodologies, component based software development methodologies and, incremental and iterative development process modeling and alike.

Chapter 1 of this thesis provided a gentle introduction to history of Software Engineering (SE) with its chronological evolution and current trends.

Chapter 2 discussed the different phases of software development with a brief introduction to different well known software development life cycle models individually with their merits, limitations.

Chapter 3 briefly introduced the various research methodologies in Software Engineering and the research methodology used in this work.

Chapter 4 discussed about the emergence of Component Based Software Development (CBSD) approaches as a remedy to economic crisis to software development and as a mechanism to manage system complexity and maintenance effort.

Chapter 5 explored the different features that any suitable software development life cycle model should bear on it. These features will be used to validate the characteristics of the BRIDGE life cycle model in the subsequent chapters.

Chapter 6 discussed our primary research work i.e. the BRIDGE software development process life cycle model in details with the exploration of its different features. The BRIDGE process model is designed based on the traditional software development paradigm incorporating the modern trends and technologies for software development those are of most concern to the industry.

Chapter 7 outlined the Agile software development philosophy with its need and importance. The Agile manifestos are also discussed briefly. Further, we discuss how the agile philosophies can also be achieved with the BRIDGE process model.

In **Chapter 8**, initially we have identified the different reasons contributing to software project failure. Thereafter, we proposed some of the remedial to the identified reasons for software project failure following the BRIDGE Process Model.

In **Chapter 9** of this thesis, we performed a comparative analysis of the BRIDGE process model in contrast to some of the well known software development lifecycle models. Next, we conclude the chapter by recommending this model to be the most suitable model for modern software development and, hence recommend this model to be used by the practitioners in software development projects to alleviate the present software development challenges i.e. software crisis.

In **Chapter 10**, we have briefly discussed a CASE tool named SRS Builder 1.0 developed following the BRIDGE process model by ourselves that is used to specify the system and customer requirements in the form of SRS document following IEEE specification.

At the end ot this thesis, in **Chapter 11**, we have summarized the entire work highlighting the achievements. Thereafter, we have concluded the work by proposing some of the directions for future work those were identified whilst conducting this work.

Abbreviations

SW	**S**oftware
HW	**H**ardware
SE	**S**oftware **E**ngineering
CBSD	**C**omponent **B**ased **S**oftware **D**evelopment
CBSE	**C**omponent **B**ased **S**oftware **E**ngineering
OOSE	**O**bject **O**riented **S**oftware **E**ngineering
OOSD	**O**bject **O**riented **S**oftware **D**evelopment
SDLC	**S**oftware **D**evelopment **L**ife **C**ycle
RAD	**R**apid **A**application **D**evelopment
JAD	**J**oint **A**application **D**evelopment
OOA	**O**bject **O**riented **A**nalysis
OOD	**O**bject **O**riented **D**esign
OOAD	**O**bject **O**riented **A**nalysis **D**esign
CASE	**C**omputer **A**ided **S**oftware **E**ngineering
SPI	**S**oftware **P**rocess **I**mprovement
SQA	**S**oftware **Q**uality **A**ssurance
UML	**U**nified **M**odeling **L**anguage
WBS	**W**ork **B**reakdown **S**tructure
SSD	**S**oftware **S**pecification **D**ocument
SRS	**S**oftware **R**equirement **S**pecification
SPM	**S**oftware **P**roject **M**anagement
SEI	**S**oftware **E**ngineering **I**nstitute
SCM	**S**oftware **C**ofiguration **M**anagement

Contents

List of Figures **xvii**

List of Tables **xviii**

1 Introduction **1**

1.1 History of Software Engineering 1

1.1.1 The Origins of Software 2

1.1.1.1 The Early Electronic Computers 2

1.1.1.2 The Early Days of Software 2

1.1.2 The Chronological Evolution of Software Engineering 5

1.1.2.1 During 1940s : The Pioneering Era 5

1.1.2.2 1945 to 1950s : Debated What Engineering Might Mean For Software! 5

1.1.2.3 1960s : The Software Crisis and Origins of Software Engineering 6

1.1.2.4 1970s Synthesis and Antithesis : Formality and Waterfall Processes 7

1.1.2.5 1985 to 1989s : No Silver Bullet 8

1.1.2.6 1990 to 1999s : Prominence of the Internet 9

1.1.2.7 2000 to Present : Lightweight Methodologies 9

1.1.3 Current Trends in Software Engineering 10

2 Some Well Known Software Development Life Cycle Models **12**

2.1 Introduction 12

2.1.1 Software Development Life Cycle or Software Process 12

2.1.2 Software Development Life Cycle Models or Process Models . . . 12

2.1.3 Software Development Approaches or Philosophies 14

2.2 Origin of Software Development Lifecycle Models 14

2.3 Typical Phases of Software Development Life Cycle 15
2.4 Introduction to Software Development Approaches or Philosophies 19
2.4.1 Iterative Software Development Philosophy 19
2.4.2 Incremental Software Development Philosophy 21
2.4.3 Agile Software Development Philosophy 23
2.5 Different Well Known SDLC Process Models 25
2.5.1 Classical Waterfall Model 26
2.5.2 Iterative Waterfall Model 29
2.5.3 Prototype Model 31
2.5.4 Spiral Model 36
2.5.5 V Model 40
2.5.6 RAD Model 42
2.5.7 RUP: Rational Unified Process Model 46
2.5.8 Evolutionary Process Model 50

3 Research Methods in Software Engineering 52
3.1 Introduction 52
3.2 What Research IS and What NOT 53
3.2.1 What research IS? 53
3.2.2 What research IS NOT? 54
3.2.3 Science Vs. Engineering 55
3.2.3.1 Science and Engineering 55
3.2.3.2 Difference between the Objectives of Science and Engineering Study 56
3.2.3.3 Software Engineering : Science or Engineering? 56
3.3 Broader View of Software Engineering Research Paradigm 58
3.3.1 **Types of Research Paradigms** 58
3.3.2 **Types of Research Problem Domain** 58
3.4 Prior Reflections on Software Engineering Research 60
3.5 Types of Software Engineering Research 61
3.5.1 Types of General Research 61
3.5.2 Types of Software Engineering Research 62
3.6 Typical Phases of Research 63
3.7 Research Strategies 64
3.7.1 Creating Research Strategies 64

3.7.2 Building Good Research Results 64
3.8 Classifications of Research Design Strategies in Software Engineering . . 64
3.8.1 Empirical research methods in software engineering 65
3.8.1.1 Controlled Experiments – high level of control and repeatable 65
3.8.1.2 Surveys – Sampling and Statistically Valid 65
3.8.1.3 Case Studies 65
3.8.2 Observational Method/Field Observation 66
3.8.3 Correlational Research 67
3.8.4 Quasi-Experimental 67
3.9 Software Engineering Research Model: Questions, Results and Validation 68
3.9.1 Software Engineering Research Questions and Types 68
3.9.2 Software Engineering Research Results and Types 69
3.9.3 Research Result Validation Techniques 70
3.9.3.1 Foundations for Acceptance of Research Results 70
3.9.3.2 Types of Validity of Empirical Studies 71
3.9.3.3 Software Engineering Research Validation Techniques . . 72
3.10 Software Engineering Research: The Road-map 73
3.10.1 Software Engineering Research Methods 73
3.10.2 Critiques of Experimental Software Engineering 74

4 Emergence of Component Based Software Engineering 76
4.1 Introduction 76
4.2 The journey of the Component Based Software Development (CBSD) Era 77
4.3 Understanding Software Components 77
4.4 Using Software Components: Component Based System Development . . 79
4.5 Objectives of Component Based Software Development 80
4.6 Impact of Component Based Software Development 81
4.7 Industrial Practices on Software Components 82
4.8 Conclusion 83

5 Investigating and Analyzing the Desired Characteristics of Software Development Lifecycle (SDLC) Models 85
5.1 Introduction 85
5.2 Objectives and Goal 86
5.3 SDLC Process Models and Objectives 86

5.4 Desired Characteristics of DLC Models 87
5.5 Conclusion 98

6 BRIDGE: A Model for Modern Software Development Process to Cater the Present Software Crisis 99
6.1 Introduction 99
6.2 Usage of Different Process Models: A Survey Report 99
6.3 Characteristics of good Software Development Process Model 100
6.4 Nature of Modern Software Projects 101
6.5 Modern Software Crisis 101
6.6 Trends in Modern Software Development 102
6.7 Reasons for failure of Traditional SDLC Models: The Shortcomings . . . 103
6.8 Need of Modified Process Model 103
6.9 BRIDGE: The Model for Modern Software Development Process 104
6.9.1 BRIDGE Process Model Description 104
6.10 Analysis of the BRIDGE Process Model 110
6.10.1 Findings from the Study of BRIDGE Model 110
6.10.2 Impact analysis of the findings from BRIDGE Model Study . . . 110
6.11 Validating the BRIDGE Model in Support of Goodness Criterion 112
6.12 Suitability of the BRIDGE Process Model 114
6.13 Limitations of the BRIDGE Process Model 114
6.14 Naming Significance: The BRIDGE 114
6.15 Conclusion 114

7 Achieving Agility through BRIDGE Process Model: An Approach to Combine the Agile and the Disciplined Software Development 115
7.1 Introduction 115
7.2 Scope of This Study 116
7.3 From Disciplined SW Development Approach towards Agile 116
7.3.1 Limitations of The Traditional Development Models 116
7.3.2 Agile Development : The Origin and its Necessity 117
7.4 Principle of Agile Development 118
7.5 Introduction to BRIDGE Process Model 120
7.6 Agile Development with BRIDGE Process Model 121
7.7 Conclusion 126

8 Identifying the Reasons for Software Project Failure and Some of their Proposed Remedial through BRIDGE Process Models 127
8.1 Introduction . . . 127
8.2 Research Goal and Objectives . . . 128
8.3 Definition of Successful and Failed Software Projects . . . 128
8.4 Project Failure Statistics . . . 128
8.5 Common Reasons for Software Project Failure and their Categorization . 130
8.6 Remedial to Project Failure Risks through BRIDGE Process Model . . . 136
8.7 Conclusion . . . 139

9 A Comparative Analysis of BRIDGE and Some Other Well Known Software Development Life Cycle Models 140
9.1 Introduction . . . 140
9.2 Software Development Approach, Process and Process Model . . . 141
9.3 Parameter Selection for Comparative Analysis . . . 141
9.4 Comparative Analysis . . . 145
9.5 Conclusion . . . 145

10 SRS BUILDER 1.0: An Upper Type CASE Tool For Requirement Specification 147
10.1 Introduction . . . 147
10.2 Computer Aided Software Engineering (CSAE) and CASE Tools . . . 148
10.3 Advantages of Using CASE Tools . . . 149
10.4 Usage of CASE Tools: A Research Report . . . 149
10.5 Limitation of CASE Tools . . . 150
10.6 Motivation for Undertaking the Research Project . . . 151
10.7 SRS BUILDER 1.0: The New Requirement Specification Tool . . . 151
10.8 Function Hierarchy Diagram of SRS BUILDER 1.0 . . . 152
10.9 Example: Sample SRS Organization Generated by SRS BUILDER 1.0 . . 152
10.10Conclusion . . . 153

11 Summary and Conclusion 156
11.1 Summary of Work . . . 156
11.2 Concluding Remarks . . . 157
11.3 Future Work . . . 158

Bibliography 159

Appendix A: List of Publications **176**

Appendix B: Copies of Some Publications **179**

List of Figures

2.1 Iterative Software Development Philosophy 20
2.2 Incremental Software Development Philosophy 22
2.3 Classical Waterfall Model 27
2.4 Iterative Waterfall Model 29
2.5 Prototype Model 32
2.6 Spiral Model 37
2.7 V (Shape) Model 40
2.8 RAD Model 43
2.9 RUP History 47
2.10 RUP (Rational Unified Process) Model 48
2.11 Evolutionary (EVO) development Process Model 50

3.1 Research (Analogy) 53

4.1 Software Components (An Abstract View) 78
4.2 Component Based System Development 79

6.1 Use of Different Process Models (in %) 100
6.2 BRIDGE Process Model 106

10.1 Use of CASE Tools in Organizations (in %) 150
10.2 FHD (Function Hierarchy Diagram) of SRS Builder 1.0 153
10.3 A sample SRS Structure Generated Using SRS Builder 1.0 (to be continued...) 153
10.4 A sample SRS Generated Using SRS Builder 1.0 (to be contd...) 154
10.5 A sample SRS Generated Using SRS Builder 1.0 (to be contd...) 154
10.6 A sample SRS Generated Using SRS Builder 1.0 155

List of Tables

8.1 Project Performance Statistics (in%) 129
8.2 Project Estimates Overrun Statistics(in%) 129

9.1 Comparison of Different SDLC Process Model 146

Chapter 1

Introduction

1.1 History of Software Engineering

It is very unfortunate that we often have very little interest in the history of any subject and many of us even thinks that the concepts and ideas are propagated and advertised as being new, but were existed decades ago perhaps under a different terminology. As said by Glass (1) that, "The most frequent mistake is the assumption that progress in those early days was slow and plodding and that not much was happening in the field.". It is worthwhile to consider the past and to investigate how terms and concepts originated and evolved day by day as we see, know and understand today.

Computer has come a long way since it was first introduced somewhere in the late 1940s. Since then it has evolved steadily throughout the ages and applications of various types of software have reached heights that were not thought to be possible. From its beginnings, writing software has evolved into a profession concerned with how best to *maximize the quality* of software and of *how to create it.* Software engineering is a rather relative term when we consider the word "engineering". The first appearance of the two words came about in the 1950s. The basic problem software engineers had was that one could not see a physical development in the software– all done virtually or on paper. So it was difficult to develop software without a proper model.

While one had, to learn that the origins of software cannot be clearly distinguished from the hardware,it will be even harder to try to predict the future of software and hardware. Will further levels of abstraction evolve? In any case, we speculate that software is still in its infancy and the origins of software will gradually come to include the present times. It is fascinating to be a part of history, especially one that has such an impact on the world.

1.1.1 The Origins of Software

1.1.1.1 The Early Electronic Computers

In 1837, Charles Babbage designed and planned to build the Analytical Engine, a programmable calculator that was supposed to be driven by programs on punch cards, enabling the machine to perform all operations possible to modern computers. The idea of programming the machine with the cards originally came from Ada Lovelace, the protégé of Babbage, who is often regarded as the worlds'first programmer. However, just like the Difference Engine, the Analytical Engine failed and was never built during Babbage's lifetime. Further, in 1837, ENIAC- the first large digital computer that could be reprogrammed was built and there was still no concept or need for portable software. In fact, there was no place where the instructions for running the machine were stored. Instead, every time the ENIAC was to compute a new problem, it had to be set up a new -– that meant re-cabeling all the hardware units manually. In the subsequent years, the manual setup was changed.
In 1944, the Mark I was developed at Harvard and in 1945 the Whirlwind at MIT was built in which punched cards were being used to determine the order of operations. It was the Harvard Mark I that spun the modern creation of software in the United States. Grace Hopper was assigned the responsibility by the Navy to create programs for the Mark I and it was in her daily routines that she saw the need for easier reuse of code and later the compiler. Her ideas were not perfect but seemed to create a series of events which would quickly lead us to the modern idea of compiles and assemblers.

1.1.1.2 The Early Days of Software

In 1804, the idea of software was first observed with the loom of Jacquard. Paper cards with punched holes controlled the weaving of the pattern on the loom, enabling more complex patterns and faster production times. His machine created the first need for software and with it the first negative reaction to programmable machines. The high-tech loom changed the weaving profession, effectively lowering the required skill set and limiting the number of people needed to operate. This loom demonstrated *three aspects of software :*

1. *logical structure*
2. *representation*
3. *interaction with the physical device*

– a concept that would loosen throughout the evolution of software. Jacquard's most significant invention was perhaps that of the punch cards – they would remain in use for well over a hundred years. Some significant uses would be Hollerith's tabulating machine for the U.S. Census, the UNIVAC and the machines built by IBM.
In 1936, Alan Turing invented the theoretical design of a computer– the *Turing Machine*, proving that the Halting problem is unsolvable (2). The Turing Machine was designed to function as a systematical working human, using only *three operations :*

1. *read* from a tape
2. *write* to a tape
3. *move the read/write head.*

With only these operations, the Turing Machine can compute anything that a functioning computer is able to calculate, and the Church Turing thesis (3) extends its capabilities by conjecturing that any function calculable in the common sense can be computed by a Turing Machine. The theoretical implications of a Turing Machine are vast, but the significant feature in this instance is the software that arose from the conception of a Turing Machine. A Turing Machine proposed that a mechanical machine would simply execute a set of instructions, which could then be easily copied or moved to a new machine. Additionally, it proposed a so called Universal Turing Machine. “A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal Turing Machine (4)”. An Universal Turing Machine that could read and simulate the behavior of other machines, given as input on the tape.
In those days, the software – mainly consisting of instructions by punch cards – was always specific to one kind of machine. The limitations of punch cards and tape reels forced extra processing time just to sort the files in an accessible manner, occupying 25% of all processing time. In order to be effective, code had to be precise and compact. Only many years later would the software evolve to become largely independent from the hardware. During these first years of software development, the study and perfecting of algorithms was quickly becoming its own subject area. Some of the best algorithms in place today were created and implemented prior to the 1970s. This, along with the constant improvements in hardware and the realization that computers can be used for more than just mathematics, created a growing need for programmers, directly creating a need for formal education in the area. Colleges began implementing computer science departments in the 1960s. The first programs seemed to focus on languages and practical

applications while contemporary programs focus more on theories and practices.
These thought of Universal Turing Machine and related experiments had many direct influences on the years to follow. Even though it took some time for them to be realized, can be seen as the earliest conceived notions on realization of compilers and interpreters and were finally implemented in 1952. Similarly, it can also be seen as the earliest conceived notions of emulators and simulators, which were only recently implemented efficiently in 1997.
In 1954, while the first languages were being created and used, Laning and Zierler developed the first assembler. In 1957, IBM released FORTAN and COBOL was popularized by the United States Government in 1960. It was during this time of innovation that the first examples of open source occurred. SHARE was a group of IBM users that joined forces to, if nothing else but share frustrating experiences. They managed to create many libraries of code, reducing the amount of redundant work between members. As the group grew, SHARE and IBM seemed to form a symbiotic relationship, in that SHARE created more profits for IBM and IBM in return placed great weight on SHARE's opinions and references. It was a great example of a developer community.
The term "software engineer" arose in 1968, when people speculated that lack of engineering approach was causing the software crisis. The NATO Science Committee sponsored two major software conferences, one in 1968 and the other in 1969. These conferences gave the initial boost required for software engineering and many mark these events as the official birth period of software engineering. Perhaps the most important occurrence in the advent of software was the unbundling of software from hardware by IBM in 1969, under pressure by the U.S. Government, and the rise of software companies. The industry slowly lost its focus on hardware, creating an expectation of reliable software. Likewise, the costs of software development started to exceed the costs for hardware.
Between 1969 and 1972, the programming language C was developed by Dennis Ritchie at AT & T's Bell Labs with other just like its direct ancestor B. Strangely enough, the initial motivation was to enable the programmers to play the video game Space Travel on a PDP11. As it happened, and almost at the same time the operating system Unics was being developed for a PDP11 machine at Bell Labs written in Assembler. The researchers soon found that the finished high level language C would enable them to make Unics. In 1973, it become necessary that the labeled UNIX to be portable to almost any other machine. So most parts of it were then rewritten for that purpose. The success of both C and UNIX was closely tied together. The fact that AT & T, being a regulated monopoly was not able to sell UNIX for profit. Along with its high portability, it led

to the fast acceptance of UNIX and C everywhere. Additionally, the source was published for a nominal fee, what led to a constant improvement of the operating system UNIX. The impressive networking, file handling and user management capabilities of UNIX influenced the early days of the Internet and many operating systems which are widely used today, such as Linux, ∗BSD or MacOS. Similarly, C was soon standardized, adapted to the PC and used by system programmers everywhere.

1.1.2 The Chronological Evolution of Software Engineering

1.1.2.1 During 1940s : The Pioneering Era

The most important development was that new computers were coming out almost every year or two, rendering existing ones obsolete. During 1940s, computer hardware was application-specific and so scientific and business tasks needed different machines. Software people had to rewrite all their programs to run on these new machines. Programmers did not have computers on their desks and had to go to the "machine room". Jobs were run by signing up for machine time or by operational staff. Jobs were run by putting punched cards for input into the machine's card reader and waiting for results to come back on the printer. Due to the need to frequently translate old software to meet the needs of new machines, high-order languages like FORTRAN, COBOL, and ALGOL were developed. Hardware vendors gave away systems software for free as hardware could not be sold without software. A few companies sold the service of building custom software but no software companies were selling packaged software. The field was so new that the idea of management by schedule was non-existent. Making predictions of a project's completion date was almost impossible.

1.1.2.2 1945 to 1950s : Debated What Engineering Might Mean For Software!

The term software engineering first appeared in the late 1950s and early 1960s. Programmers have always known about civil, electrical, and computer engineering and debated what engineering might mean for software. During 1950s people believed understanding was, "Engineer software like you engineer hardware"(5). First, software was much easier to modify than was hardware, and it did not require expensive production lines to make product copies. One changed the program once and then reloaded the same bit pattern onto another computer, rather than having to individually change the configuration of each copy of the hardware. This ease of modification led many people and organizations to adopt a "code and fix"approach to software development, as compared to the

exhaustive "Critical Design Reviews" that hardware engineers performed before committing to production lines and bending metal (measure twice, cut once). Many software applications became more people-intensive than hardware-intensive; even SAGE became more dominated by psychologists addressing human-computer interaction issues than by radar engineers. Another software difference was that software did not wear out. Thus, software reliability could only imperfectly be estimated by hardware reliability models, and software maintenance was a much different activity than hardware maintenance. By the 1960s, however, people were finding out that software phenomenology differed from hardware phenomenology in significant ways.

1.1.2.3 1960s : The Software Crisis and Origins of Software Engineering

During 1960s, the code-and-fix was the primary software development approach followed by the industry. But during the 1960s, 70s and 80s brought about the so called *software crisis*. Software was invisible, it did not weigh anything, but it cost a lot. This time frame became a very bumpy road for software developers and engineers. During this period many of the problems in software development were highlighted. Initially the *software crisis* was defined in terms of productivity, but later it turned out to be defined in terms of quality. Many software projects ran over budget or over schedule.
Software engineering has come a long way since the 1960s and the first attempts to make our field into an engineering discipline. The philosophy during 1960s was "think outside the box". Repetitive engineering would never have created the Arpanet or Engelbart's mouse-and-windows GUI. Have some fun prototyping; it is generally low-risk and frequently high reward. Respect software's differences. You cannot speed up its development indefinitely. Since it is invisible, you need to find good ways to make it visible and meaningful to different stakeholders. Avoid cowboy programming. The last-minute all-nighter frequently does not work, and the patches get ugly fast.
This situation led the NATO Science Committee (5) to convene two landmark Software Engineering conferences in 1968 (Garmisch, Germany) and 1969, attended by many of the leading researcher and practitioners in the field. In these conferences the difficulties and pitfalls of designing complex systems were frankly discussed. These particular conferences gave the field its initial boost. Many believe these conferences marked the official start of the profession of software engineering.
Software generally had many more states, modes, and paths to test, making its specifications much more difficult. Winston Royce, in his classic 1970 paper, said, *In order to procure a $5 million hardware device, I would expect a 30 page specification would*

provide adequate detail to control the procurement. In order to procure $5 million worth of software, a 1500 page specification is about right in order to achieve comparable control (5). It was hard to tell whether it was on schedule or not, and if you added more people to bring it back on schedule, it just got later, as Fred Brooks explained in the "Mythical Man-Month"(5). A few projects caused loss of life (5). Eventually the hard work of many software companies paid off and reviled the path towards a brighter future of software engineering.

1.1.2.4 1970s Synthesis and Antithesis : Formality and Waterfall Processes

The main reaction to the 1960s code-and-fix approach involved processes in which coding was more carefully organized and was preceded by design, and design was preceded by requirements engineering. *The philosophy during 1970s was to eliminate errors early.* Even better, prevent them in the future via root cause analysis. Determine the system's purpose. Without a clear shared vision, you are likely to get chaos and disappointment. Goal-question metric is another version of this. Avoid Top-down development and reductionism. COTS, reuse, IKIWISI, rapid changes and emergent requirements make this increasingly unrealistic for most applications.

More careful organization of code was exemplified by Dijkstra's (5) famous letter to ACM Communications, "Go To Statement Considered Harmful". The Bohm-Jacopini result showing that sequential programs could always be constructed without goto's led to the Structured Programming movement. This movement had two primary branches. One was a "formal methods"branch that focused on program correctness, either by mathematical proof (5) or by construction via a "programming calculus". The other branch was a less formal mix of technical and management methods, "top-down structured programming with chief programmer teams", pioneered by Mills and highlighted by the successful New York Times application led by Baker.

The success of structured programming led to many other "structured "approaches applied to software design. Principles of modularity were strengthened by Constantine's concepts of coupling (the degree of interdependency between two modules- to be minimized) and cohesion (the degree of functional strength of a module - to be maximized), by Parnas's increasingly strong techniques of information hiding, and by abstract data types. A number of tools and methods (5) employing structured concepts were developed, such as structured design; Jackson's structured design and programming, emphasizing data considerations and Structured Program Design Language.

Requirements-driven processes were well established in the 1956 SAGE process model,

but a stronger synthesis of the 1950s paradigm and the 1960s crafting paradigm was provided by Royce's version of the "waterfall "model (5). During 1970s Software engineering was mostly qualitative.

1.1.2.5 1985 to 1989s : No Silver Bullet

The cost of owning and maintaining software in the 1980s was twice as expensive as developing the software. During the 1990s, the cost of ownership and maintenance increased by 30% over that of 1980s. In 1995, statistics showed that half of surveyed development projects were operational, but were not considered successful. The average software project overshoots its schedule by half. Three-quarters of all large software products delivered to the customer are failures that are either not used at all, or do not meet the customer's requirements.
For decades, solving the software crisis was paramount to researchers and companies producing software tools. Almost every new technology and practice from the 1970s to the 1990s was trumpeted as a *silver bullet* to solve the software crisis. Tools, discipline, formal methods, process and professionalism were touted as silver bullets. Debate about silver bullets raged over the following decade. Advocates for Ada, components and processes continued arguing for years that their favorite technology would be a silver bullet. Skeptics disagreed. The search for a single key to success never worked. In 1986, Fred Brooks (6) published his No Silver Bullet article, arguing that no individual technology or practice would ever make a 10-fold improvement in productivity within 10 years. Eventually, almost everyone accepted that no silver bullet would ever be found. Yet, claims about silver bullets pop up now and again even today. Some interpret no silver bullet to mean that software engineering failed. Others interpret no silver bullet as proof that software engineering has finally matured and recognized that projects succeed due to hard work. However, Brooks goes on to say, "We will surely make substantial progress over the next 40 years; an order of magnitude over 40 years is hardly magical ...". All known technologies and practices have only made incremental improvements to productivity and quality. Yet, there are no silver bullets for any other profession, either. However, it could also be said that there are, in fact, a range of silver bullets today, including lightweight methodologies spreadsheet calculators, customized browsers, in-site search engines, database report generators, integrated design-test coding-editors with memory/differences/undo and specialty organizations that generate niche software at a fraction of the cost of totally customized website development. Nevertheless, the field of software engineering appears too complex and diverse for a single "silver bullet"to

improve most issues, and each issue accounts for only a small portion of all software problems.

1.1.2.6 1990 to 1999s : Prominence of the Internet

With the start of the 1900s came as never before seen phenomenon called the "Internet"and World Wide Web (WWW) brought out opportunities like never before. The rise of the Internet led to very rapid growth in the demand for international information display/e-mail systems on the WWW. Programmers were required to handle illustrations, maps, photographs, other images in addition to simple animation at a rate never before seen with few well known methods to optimize image display or storage. The growth of browser usage running on the HTML language changed the way in which information display and retrieval was organized. The widespread network connections led to the growth and prevention of international computer viruses on Windows computers. The vast proliferation of spam e-mail became a major design issue in e-mail systems along with flooding communication channels and requiring semi automated prescreening. Keyword search systems evolved into web based search engines, and many software systems had to be redesigned for international searching depending on Search Engine Optimization (SEO) techniques. Human natural language translation systems were needed to attempt to translate the information flow in multiple foreign languages with many software systems being designed for multi-language usage based on design concepts from human translators. Typical computer user bases went from hundreds or thousands of users to many-millions of international users.

1.1.2.7 2000 to Present : Lightweight Methodologies

With the expanding demand for software in many smaller organizations, the need for inexpensive software solutions led to the growth of simpler and faster methodologies that developed running software from requirements to deployment in much quicker and easier way. The use of rapid prototyping evolved to entire lightweight methodologies such as Extreme Programming (XP), which attempted to simplify many areas of software engineering, including requirements gathering and reliability testing for the growing vast number of small software systems. Very large software systems still used heavily documented methodologies with many volumes in the documentation set. However, smaller systems had a simpler and faster alternative approach to managing the development and maintenance of software calculations and algorithms, information storage or retrieval, and display.

The 21^{st} century brought out some of the best programmers and programs of all time. Software became user friendly and easy to use. Programmers were looking for easier and better ways to write down codes. Life for both the software engineer and the end user became much, much easier.

1.1.3 Current Trends in Software Engineering

Software engineering is relatively a young discipline and is still developing. The directions in which software engineering is developing recently includes :

- **Component Based Software Development (CBSD) :** Component Based Software Development aims to construct complex software systems by means of integrating reusable software components. This approach promises to alleviate the software crisis at great extents following the principle of abstraction.
- **Agile Philosophy :** Agile software development guides software development projects that evolve rapidly with changing expectations and competitive markets. Proponents of this method believe that heavy and document driven processes (like TickIT, CMM and ISO 9000) are fading in importance. Some people believe that companies and agencies export many of the jobs that can be guided by heavyweight processes. Agile philosophy are incorporated in the extreme programming, scrum and lean software development.
- **Aspects Orientation :** Aspects help software engineers'to deal with quality attributes by providing tools to add or remove boilerplate code from many areas in the source code. Aspects describe how all objects or functions should behave in particular circumstances. For example, aspects can add debugging, logging or locking control into all objects of particular types. Researchers are currently working to understand how to use aspects to design general purpose code. Related concepts include generative programming and templates.
- **Experimental Software Engineering :** Experimental software engineering is a branch of software engineering interested in devising experiments on software, in collecting data from the experiments and in devising laws and theories from this data. Proponents of this method advocate that the nature of software is such that we can advance the knowledge on software through experiments only.

- **Model driven Approach :** Model driven design develops textual and graphical models as primary design artifacts. Development tools are available that use model transformation and code generation to generate well organized code fragments that serve as a basis for producing complete applications.
- **Software Product Lines :** Software product lines are a systematic way to produce families of software systems instead of creating a succession of completely individual products. This method emphasizes extensive, systematic and formal code reuse to try to industrialize the software development process.

The Future of Software Engineering (5) conference (FOSE) held at ICSE 2000 documented the state of the art of SE and listed many problems to be solved over the next decade. The FOSE tracks at the ICSE 2000 and the ICSE 2007 conferences also help identify the state of the art in software engineering.
Software has formed an everlasting place in society, because humanity could never return to life without it. Some people even hypothesize that we are already living in the age of artificial intelligence because humans are now capable of doing things not normally humanly possible.

Chapter 2

Some Well Known Software Development Life Cycle Models

2.1 Introduction

It is really tough to draw a sharp line between software development approaches and Software Development Life Cycle (SDLC) models. In many literature of software engineering, these terms are even used interchangeably. So, before we begin the details discussion of the topic, let us somehow draw the boundary line between software development approaches and SDLC process models. Defining these two terms are beyond the scope of this chapter, however we just try to explain the both only to establish the differences from our point of view.

2.1.1 Software Development Life Cycle or Software Process

The terms Software Development Life Cycle (SDLC) and software process are used interchangeably or in conjunction. A SDLC spans over the time period from the concept development to the product retirement. Software development life cycle involves the step-by-step activities in the development of a software product. The whole process is generally classified into a set of steps and a specific operation will be carried out in each of the steps. SW development process or simply process typically defines the set steps to be carried out during the development of the system.

2.1.2 Software Development Life Cycle Models or Process Models

The SDLC process model typically depicts the fashions in which the SW process to be carried out i.e. which steps to be done before or after another step. A software develop-

ment process model is an approach to the SDLC that describes the sequence of steps to be followed while developing software projects (7, 8). In general all the process models do cover all distinct phases defined by SW process, but in different manner or sequence– which makes one process model differ from the other.

a. Purposes for articulating software life cycle models

There are a variety of purposes for articulating software life cycle (SDLC) models. SDLC model serve as a (9, 10, 11) :

- guideline to organize, plan, staff, budget, schedule and manage software project work over organizational time, space and computing environments
- prescriptive outline for what documents to produce for delivery to client
- basis for determining what software engineering tools and methodologies will be most appropriate to support different life cycle activities
- framework for analyzing or estimating patterns of resource allocation and consumption during the software life cycle
- basis for conducting empirical studies to determine what affects software productivity, cost, and overall quality

b. Types of SDLC Process Models

A software life cycle model is basically the characterization of how software is or should be developed. The SDLC process model can be either Perspective of Descriptive.

1. ***Prescriptive* SDLC Process Model :** A *prescriptive model* prescribes how a new software system should be developed and are used as guidelines or frameworks to organize and structure how software development activities should be performed in specific order. Typically, it is easier and more common to articulate a prescriptive life cycle model for how software systems should be developed. This is possible since most such models are intuitive or well reasoned. This means that many idiosyncratic details that describe how a software system is built in practice can be ignored, generalized or deferred for later consideration. This, of course, should raise concern for the relative validity and robustness of such life cycle models when developing different kinds of application systems in different kinds of development settings, using different programming languages with differentially skilled staff etc. However, prescriptive models are also used to package the development tasks and

techniques for using a given set of software engineering tools or environment during a development project.

2. ***Descriptive* SDLC Process Model :** A *descriptive model* describes the history and characterize how particular software systems are actually developed in specific settings. Descriptive models may be used as the basis for understanding and improving software development processes or for building empirically grounded prescriptive models (12). As such, they are less common and more difficult to articulate for obvious reasons:
   - one must observe or collect data throughout the life cycle of a software system, a period of elapsed time often measured in years.
   - also, descriptive models are specific to the systems observed and only can be generalize through systematic comparative analysis.

Therefore, this suggests the prescriptive software life cycle models will dominate attention until a sufficient base of observational data is available to articulate empirically grounded descriptive life cycle models.

2.1.3 Software Development Approaches or Philosophies

Several software development philosophies have been proposed by different authors over the time to address the challenges in software development. The most common software development philosophies are like iterative, incremental, agile, evolutionary and component based development philosophies. These software development philosophies can be implemented following other process models i.e. Waterfall, RAD, Spiral, Prototyping or alike for better and optimal results. These software development philosophies are discussed briefly in the subsequent sections.

2.2 Origin of Software Development Lifecycle Models

In early days, software engineering approach was *ad hoc*. Around 1970s, introduction of *structured programming* gave a formal shift in software engineering from the ad hoc to a systematic approach. Then around 1980s, introduction of *object oriented programming* with some advancement explores new areas in software engineering. In recent dates, with the introduction of *Component Based Software Development (CBSD)*, the industry

is moving in a new direction. These day's software systems are more complex as compared to those of early. These complex, high quality software systems are built efficiently using component based approach in a shorter time. The importance of component based development lies in its efficiency. In the remaining part of this chapter the term *component* and *software components* will be used interchangeably.
Explicit models of software evolution date back to the earliest projects developing large software systems in the 1950s and 1960s (13, 14). Overall, the apparent purpose of these early software life cycle models was to provide a conceptual scheme for rationally managing the development of software systems. Such a scheme could therefore serve as a basis for planning, organizing, staffing, coordinating, budgeting, and directing software development activities.

2.3 Typical Phases of Software Development Life Cycle

The typical steps of any Software (SW) Development Life Cycle Models are as follows :

1. Project Planning
2. Risk Analysis
3. Software Requirement Gathering, Analysis, and Specification
4. Feasibility Study
5. System Analysis and Design
6. Implementation or Code Generation
7. Testing
8. System Deployment and Operation
9. Maintenance and Support
10. Customer Evaluation

Each of the steps has its own importance and plays a significant role during the product development. As description of each of the steps can give a better understanding, we briefly discussed these phases below :

1. **Project Planning :** This is the first and foremost important stage in the development. The basic motive is to plan the total project and to estimate the merits and demerits of the project. The planning phase includes the definition of the intended system, development of the project plan and parallel management of the plan throughout the proceedings. A good and matured plan can create a very good initiative and can positively affect the complete project. In this phase, the objectives, alternatives and constraints of the project are determined and documented. The objectives and other specifications are fixed in order to decide which strategies or approaches to follow during the project life cycle.

2. **Risk Analysis :** In this phase, all possible alternatives which can help in developing a cost effective project are analyzed and strategies are decided so as to use them. This phase has been added specially in order to identify and resolve all the possible risks in the project development. If risks indicate any kind of uncertainty in requirements, prototyping may be used to proceed with the available data and find out a possible solution in order to deal with the potential changes in the requirements.

3. **Software Requirement Gathering, Analysis, and Specification :** The essential purpose of this phase is to find the need and to define the problem that needs to be solved. As software is always of a large system, before software companies start working on any project, they do an in-depth analysis of client requirements for the project. This helps to suggest the best solution which will suit customers'current requirements and also which is capable of scaling to accommodate the future requirements as well. System is the basic and very critical requirement for the existence of software in any entity. So if the system is not in place, the system should be engineered and put in place. In some cases, to extract the maximum output, the system should be re-engineered and spruced up. The project work begins by establishing the requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when the software must interface with other elements such as hardware, people and other resources. The *requirement gathering* process is intensified and focused specially on software. To understand the nature of the program(s) to be built, the system engineer or system analyst must understand the information domain for the software, as well as required functions, behavior, performance and interfacing. In this phase, the development team visits the customer and studies their system. They investigate

the need for possible software automation in the given system. The main aim of the *analysis phase* is to perform statistics and requirements gathering. Based on the analysis of the project and due to the influence of the results of the planning phase, the requirements for the project are decided and gathered. Once the ideal system is engineered or tuned, the development team studies the software requirement for the system.

In the *requirement specification* activity, after the requirements for the project are analyzed, they are prioritized and made ready by specifying in a document called *System Requirement Specification* (SRS) for further use.

4. **Feasibility Study:** In this phase, the various feasibilities i.e. economic, operational, technical and environmental feasibilities of the project are analyzed. By the end of the feasibility study, the team furnishes a document that holds the different specific recommendations for the candidate system. It also includes the personnel assignments, costs, project schedule, target dates etc. Most of the developers have the habit of developing a prototype of the entire software and represent the software as a miniature model. The flaws, both technical and design, can be found and removed and the entire process can be redesigned.

5. **System Analysis and Design :** Once the analysis is over, the design phase begins. This is the first phase when a project kicks-off. The aim is to create the architecture of the total system following a modular approach. In this phase, the overall structure and its nuances are defined for the software. This is one of the important stages of the process and serves to be a benchmark stage since the errors performed until this stage and during this stage can be rectified here. Any glitch in the design phase could be very expensive to solve in the later stage of the software development. During this phase, software companies come up with the designs for the front-end as well as a scalable and robust application framework including the database design. In terms of the client/server technology, the *number of tiers needed* for the package *architecture*, the *database design*, the *data structure design* etc... are all defined in this phase. Thus a software development model is created and the logical system of the product is developed in this phase.

6. **Implementation/Code Generation :** Once the design is finalized, reviewed and approved then the design must be translated into a machine readable form using the appropriate programming languages like C, C++, Pascal, Java, J2EE, etc. as per the project type and its requirements. The application development actually

starts now.

The implementation or code generation step performs this task. If the design is performed in a detailed manner, code generation can be accomplished without much complication. Programming tools like editors, compilers, interpreters, debuggers etc. are used to generate the code.

7. **Testing :** Once the code is generated, the software testing begins. The testing phase is one of the final stages of the development process and this is the phase where the final adjustments are made before presenting the completely developed software to the end user. In general, the testers encounter the problem of removing the logical errors and bugs. Unit testing, integration testing and system testing are done at different milestones of the project to ensure a bug free delivery. The test conditions which are decided in the analysis phase are applied to the system and if the output obtained is equal to the intended output, it means that the software is ready to be provided to the user. Different testing tools and methodologies are already available to unravel the bugs that were committed during the previous phases. Some companies build their own testing tools that are tailor made for their own development operations. Adherence to various programming and testing standards is strictly followed to ensure the code quality and performance.

8. **System Deployment and Operation :** After the completion of coding and testing, the application is deployed in the actual operational environment and it goes live. Software companies also provide different kind of application support to keep your application always on the go.

9. **Maintenance and Support :** The software will definitely undergo change once it is delivered to the customer due to many reasons. The software should be developed to accommodate changes that could happen during the post implementation period. Change could happen because of some unexpected input values into the system. Any updates required to the application, once it goes live, can be done as part of project maintenance work. Further, the changes in the system could directly affect the software operations. Thus, the toughest job is encountered in the maintenance phase which normally accounts for the highest amount of money. The maintenance team is decided such that they monitor on the change in organization of the software and report to the developers, in case a need arises. The information desk is also provided if required in this phase to maintain the relationship between the user and the creator.

10. **Customer Evaluation :** In this phase, developed product is passed on to the customer in order to receive comments and suggestions which can help in identifying and resolving potential problems/errors in the software developed. This feedback report may be used by the organizations for process improvement and knowledge enhancements through learning from their past mistakes.

Any process model may further combine one or more of these phases or even may split a single phase in to multiple sub-phases or individual phases depending on the project need and by means of this different process models differs among themselves.

2.4 Introduction to Software Development Approaches or Philosophies

Several software development philosophies have been proposed by different authors over the time to address the challenges in software development. The most common software development philosophies are like Iterative development, incremental development, agile development, component based development philosophies etc. These software development philosophies may be followed together with some Software Development Lifecycle Models (SDLC) for better and optimum results. In this section we shall outline these philosophies briefly.

2.4.1 Iterative Software Development Philosophy

An iterative Software Development does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software which can then be reviewed in order to identify further requirements. This process is then repeated, producing a new version of the software at each iteration. The iterative software development philosophy can be likened to produce software by successive approximation. The key to successful use of an iterative software development lifecycle is rigorous validation of requirements and verification of each version of the software against those requirements within each iteration of the model. The following Figure 2.1 depicts the iterative software development philosophy :

Features of Iterative Software Development Philosophy :

The features of iterative approach those made it generally superior to a linear approach are:

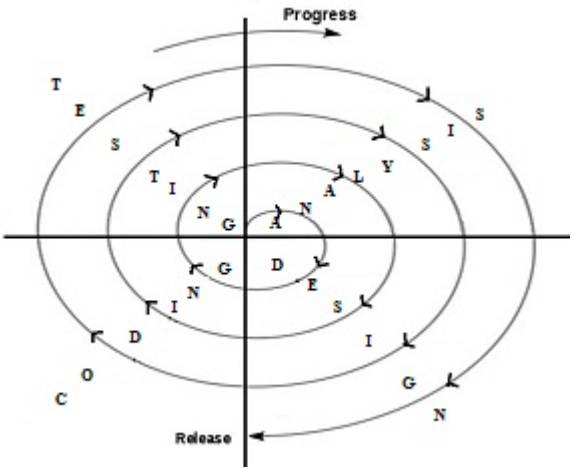


Figure 2.1: Iterative Software Development Philosophy

- **Requirements Change Consideration :** The truth is that requirements usually change over the lise span of a system. Iterative development lets one to take into account changing requirements. Changing requirements and requirements "creep"have always been primary sources of project trouble, which lead to late delivery, missed schedules, unsatisfied customers and frustrated developers.

- **Progressive Integration :** In iterative development, integration is not one "big bang"at the end; instead, elements are integrated progressively. The iterative approach is almost a process of continuous integration.

- **Early Risk Mitigation :** The iterative approach lets mitigate risks earlier because integration is generally the only time that risks are discovered or addressed. As you unroll the early iterations, you go through all process components, exercising many aspects of the project, such as tools, off-the-shelf software, people skills, and so on. Perceived risks will prove not to be risks, new, unsuspected risks will be revealed.

- **Making Tactical Changes :** It provides management with a means of making tactical changes to the product for whatever reason, for example, to compete with existing products. You can decide to release a product early with reduced functionality to counter a move by a competitor or one can adopt another vendor for a given technology.

- **Reuse :** It facilitates reuse because it is easy to identify common parts as they are partially designed or implemented instead of identifying all commonality in the

beginning before anything has been designed or implemented which is typically difficult. Design reviews in early iterations allow architects to identify unsuspected potential reuse and then develop, and mature common code for it in subsequent iterations.

- **Robust :** It results in a very robust architecture because you correct errors over several iterations. Flaws are detected even in the early iterations as the product moves beyond inception into elaboration rather than in one massive testing phase at the end. Performance bottlenecks are discovered at a time when they can still be addressed instead of creating panic on the eve of delivery.

- **Progressive Learning :** Developers can learn along the way, and their various abilities and specialties are employed more fully during the entire lifecycle. Testers start testing early, technical writers write early, and so on. In a non-iterative development, the same people would be waiting around to begin their work, making plan after plan but not making concrete progress. What can a tester test when the product consists only of three feet of design documentation on a shelf? Training needs or the need for additional people is spotted early during assessment reviews.

- **Improved Process :** The development process itself can be improved and refined along the way. The assessment at the end of each iteration not only looks at the status of the project from a product or schedule perspective but also analyzes what should be changed in the organization and in the process to make it perform better in the next iteration.

2.4.2 Incremental Software Development Philosophy

The incremental software development philosophy (as depicted in Figure 2.2) combines elements of the linear sequential model in a staggered fashion as calendar time progresses with the iterative philosophy of prototyping repeatedly. Each linear sequence produces a deliverable "increment"of the software. For example, the ATM system of any bank developed using the incremental paradigm might deliver money withdrawal and balance inquiry capabilities in the first increment; money transfer capability in the second increment; and other capabilities in the subsequent increments.

When an incremental model is used, the first increment is often a core product. The basic requirements are addressed, but many known and unknown supplementary features

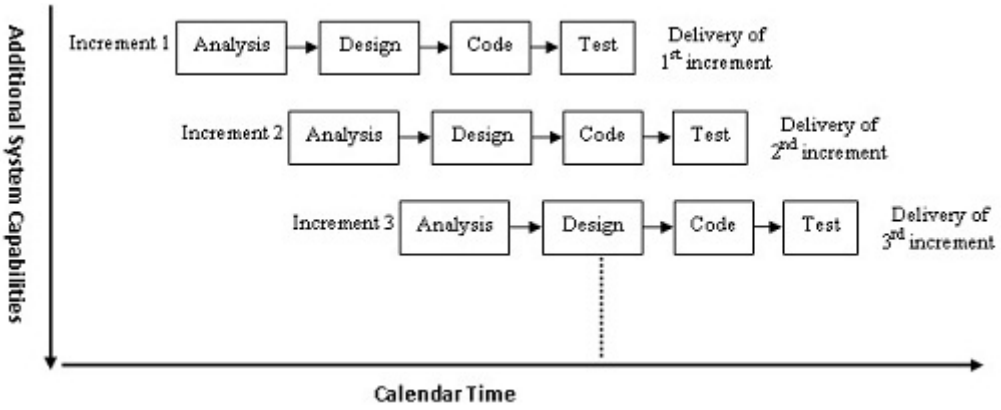


Figure 2.2: Incremental Software Development Philosophy

remain undelivered. The core product is used by the customer.

Features of Incremental Software Development Philosophy :
The incremental philosophy has the following features:

- Early increments can be developed with few people
- It combines iterative nature of prototyping model and linear nature of Linear Sequential Model
- Less number of people are required
- Improves product quality
- The system can be designed in such a manner that it can be delivered into pieces
- Increments are developed one after the other, after feedback has been received from the user
- Since each increment is simpler than the original system, it is easier to predict resources needed to accomplish the development task within acceptable accuracy bounds
- Increments can be planned to manage technical risks
- It can be applied to those projects which have independent modules

2.4.3 Agile Software Development Philosophy

Agile software development is a concept, a philosophy and a methodology which evolved during 1990s as an answer to the long growing frustrations of the waterfall SDLC concepts. The term promotes an iterative approach to software development using shorter and lightweight development cycles and some different deliverables. Agile software development is a philosophy for managing software projects and teams. Agile represents a group of software engineering methodologies which promise to deliver increased productivity, quality and project success rate overall in software development projects.

Features of Agile Software Development Philosophy :

The features of Agile software development philosophy are as follows :

- **Modularity :** Modularity allows a process to be broken into components called activities capable of transforming the vision of the software system into reality.
- **Iterative :** Agile software processes focus on short iterations or cycles typically started and completed in a matter of weeks. Within each cycle, a certain set of activities is completed. However, a single cycle will probably not be enough to complete the entire project. Therefore, the short cycle is repeated many times to refine the deliverables.
- **Time-Bound :** Time-boxing having a fixed time limit is a popular technique for bounding an iteration (15). This technique forces trade-offs to be made in the elements of the project and also encourages optimization of the agile process. One thing that time boxes are not allowed to be used for is to pressure project members into making poor decisions, working overtime, and cutting corners on quality (16).
- **Parsimony :** We must implement and test until the system works as it is intended to. These activities are not negotiable. Until someone radically changes the way that we create software, there will be a certain set of mandatory activities necessary to deliver systems. Agile software processes focus on parsimony. That is, they require a minimal number of activities necessary to mitigate risks and achieve their goals. By minimizing the number of activities, they allow developers to deliver systems against an aggressive schedule maintaining some semblance of a normal life.
- **Adaptive :** During an iteration, new risks may be exposed which require some activities that were not planned. The agile process adapts the process to attack

these new found risks. If the goal cannot be achieved using the activities planned during the iteration, new activities can be added to allow the goal to be reached. Similarly, activities may be discarded if the risks turn out to be ungrounded.

- **Incremental :** An agile process does not try to build the entire system at once. Instead, it partitions the nontrivial system into increments which may be developed in parallel at different times and at different rates. We unit test each increment independently. When an increment is completed and tested, it is integrated into the system. Each increment may require several iterations to complete.

- **Convergent :** Convergence states that we are actively attacking all of the risks worth attacking. As a result, the system becomes closer to the reality that we seek with each iteration. As risks are being proactively attacked, the system is being delivered in increments. We are doing everything within our power to ensure success in the most rapid fashion. Convergence is taken for granted in most software development processes and subsequently is often assumed. However, if risks such as "excessive feature creep"are not addressed early in the project, the project can spiral out of control. Excessive feature creep is not the same as changing requirements. Requirements will change over the course of a project. These changes are handled by the iterative and adaptive characteristics of the agile process. If requirements do change, there should be an activity in the process to capture these changes. However, a user can change his mind more rapidly than software can developed. If the requirements are not converging things are probably out of control.

- **People Oriented :** Small teams are a central theme of agile processes. Agile process empower developers by creating small teams around an increment (17). Even large projects using agile software processes divide their members into smaller teams. This gives people the sense that they are not isolated elements in a large organization (17). Increments provide the ideal deliverable for these teams.

- **Collaborative :** Communication is a vital part of any software development project. When a project is developed in pieces, understanding how the pieces fit together is vital to creating the finished product. There is more to integration than simple communication. Quickly integrating a large project while increments are being developed in parallel, requires collaboration. Agile processes foster communication among team members.

- **Complimentary :** Complimenting is an aspect of good process design that utilizes downstream activities to validate and enhance the outputs of earlier activities. Complimentary activities are activities that work together to produce a better result than they would individually. For example, the activities, "write user stories", "create acceptance tests"and "estimate the user story"are complimentary because they produce descriptions of functionality that can be tested and written within the scope on an iteration. These three activities are part of Extreme Programming.

2.5 Different Well Known SDLC Process Models

Many people have proposed different software development process models. Many of them are quite same in different aspects while other differs. Here we just consider some well known SDLC process models enlisted below :

- Classical Waterfall Model
- Iterative Waterfall Model
- Prototype Model
- Spiral Model
- V-Model
- RAD Model
- RUP Model
- Evolutionary Model

The details discussion of these SDLC model is beyond the scope of this thesis, but we just highlight the features of these models which are important for our considerations in the coming sections. The readers may follow the references for further detail discussion of these process models (18, 19, 20).

2.5.1 Classical Waterfall Model

History of Classical Waterfall Model

The first formal description of the classical waterfall model (shown in Figure 2.3) or simply waterfall model is often cited as a 1970 article by Winston W. Royce(21), though he did not use the term "waterfall"in his article. As an initial concept Royce presented this model as an example of a flawed, nonworking model. The phrase "waterfall model"quickly came to refer not to Royce's final, iterative design, but rather to his purely sequentially ordered model. Royce (21) started his 1970 article "Managing the development of large software systems"with a statement about the origin of his ideas : *I am going to describe my personal views about managing large software developments. I have had various assignments during the past nine years, mostly concerned with the development of software packages for spacecraft mission planning, commanding and post-flight analysis. In these assignments I have experienced different degrees of success with respect to arriving at an operational state, on time, and within costs. I have become prejudiced by my experiences and I am going to relate some of these prejudices in this presentation.* The waterfall development model originates in the manufacturing and construction industries : highly structured physical environments in which after-the-fact changes are prohibitively costly, if not impossible. Since no formal software development methodologies existed at the time, this hardware oriented model was simply adapted for software development. Waterfall is a software development model with strictly one Iteration/phase. In this process model, development proceeds sequentially through the phases : requirements analysis, design, coding, testing, integration, and maintenance (14).

Features of Classical Waterfall Model

Waterfall approach was first Process Model to be introduced and followed widely in Software Engineering to ensure success of the project. In this approach, the whole process of software development is divided into separate process phases. The waterfall model is a sequential design process in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Requirement and Analysis, Design, Code, Testing, Implementation and Maintenance(14) in a strict, planned sequence.

Advantages of Classical Waterfall Model

The advantages of waterfall model are :

- employs a systematic, orthodox method of project development and delivery
- simple to understand and use

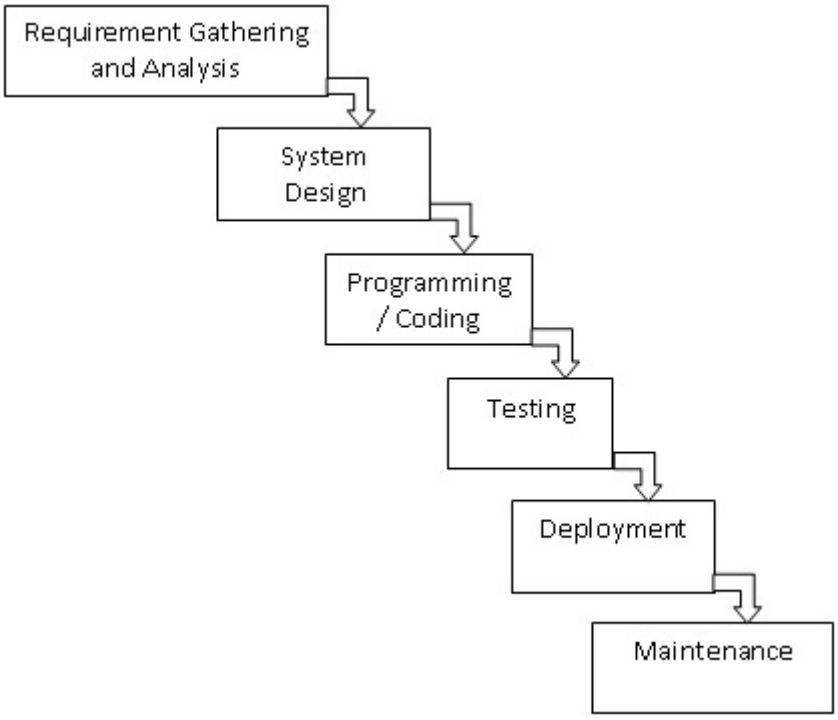


Figure 2.3: Classical Waterfall Model

- clearly defined stages
- clear project objectives
- strict sign-off requirements -stable project requirements
- progress of system is measurable through well understood milestones
- this model is extremely easy to understand and therefore, is implemented at various project management levels and in a number of fields
- easy to arrange tasks
- process and results are well documented
- each phase has specific deliverable and a review
- the waterfall methodology is easy on the manager
- customers/End users already know about it
- it allows for departmentalization and managerial control. Each phase of development proceeds in strict order, without any overlapping or iterative steps

- provides a disciplined and structured approach which makes it easy to keep projects under control
- limits the amount of cross team interaction needed during development

Disadvantages of Classical Waterfall Model

The disadvantages of waterfall model are :

- it does not allow for much reflection or revision i.e. difficulty in responding to changes
- time consuming
- customers have unreasonable time lines and expectations
- communication gaps exist between customers, engineers and project managers
- this methodology can be grueling for developers as oversights and flawed design work can seriously affect the budgeted costs and final launch date
- debugging can be complicated
- leaves no room for feedback anywhere in the process except at the end
- the methodology can give the false expectation of predictability and the reality of cost and date overruns when applied to the wrong situations
- it doesn't support iterative methodology
- being a strictly sequential model, jumping back and forth between two or more phases is impossible
- bugs and errors in the code cannot be discovered until and unless the testing phase is reached leading to wastage of time and other precious resources
- customers don't (really) know at the beginning what they want but any scope for adjustment during the life cycle can end a project
- requirements change during the course of the project is difficult
- no working software is produced until late in the life cycle– the deliverable is a onetime at the end of the complete development

- risk and uncertainty is high with this process model
- users can only judge quality at the end

Suitability of Classical Waterfall Model

- works well for projects where requirements are well understood but not suitable when the project requirements are dynamic or constantly changing
- works well when quality is more important than cost and schedule
- not suitable for complex projects

2.5.2 Iterative Waterfall Model

The primary limitation of the classical waterfall model is that it doesn't allow going back to its previous phases from any other phase. Hence, if any error or issue is detected at any phase which requires making necessary changes to its previous or earlier phase deliverables– is simply impossible with classical water fall model. In Iterative Waterfall Model (Figure 2.4) this particular limitations is overcome by adding backward path to its previous phases from any particular phase.

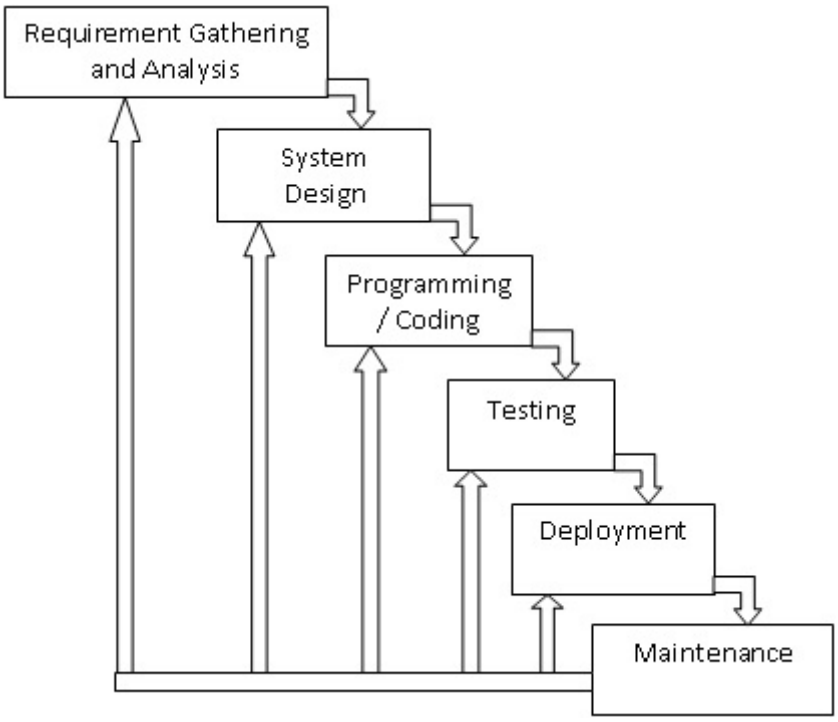


Figure 2.4: Iterative Waterfall Model

Advantages of Iterative Waterfall Model

In addition to the advantages of Classical Waterfall Model, Iterative Waterfall Model provides the following advantages :

- allows for much reflection or revision– although an application is in the testing stage, it is possible to go back and change something that was not well thought out in the concept stage facilitating responding to changes
- allows iteration and flexibility
- carries less risk than a classical Waterfall Model
- one can find and correct defects over several iterations producing a robust architecture and high quality application
- provides many opportunities during the development to learn from earlier mistakes and improve developer skills from one iteration to another ensuring team members to learn along the way
- produces better quality products

Disadvantages of Iterative Waterfall Model

The disadvantages of iterative waterfall model are :

- time consuming
- customers have unreasonable timelines and expectations
- communication gaps exist between customers, engineers and project managers
- when applied to the wrong situations the methodology can give the false expectation of predictability and the reality of cost and date overruns.
- users can only judge quality at the end
- very risky, since one process cannot start before finishing the other

Suitability of Iterative Waterfall Model

The iterative waterfall model is suitable for the types of projects when the project understanding and system requirements are not very clear at the beginning of the project.

2.5.3 Prototype Model

Software Prototyping

During 1960s and 1970s monolithic development cycle of building the entire program first and then working out any inconsistencies between design and implementation were followed. That led to higher software costs and poor estimates of time and cost. The monolithic approach has been dubbed the "Slaying the (software) Dragon"technique, since it assumes that the software designer and developer is a single hero who has to slay the entire dragon alone.

A prototype is a working model that is functionally equivalent to a component of the product which is not based on strict planning, but is an early approximation of the final product or software system and acts as a sample to test the process. From this sample we learn and try to build a better final product. A prototype typically implements only a small subset of the features of the eventual program, and the implementation may be completely different from that of the eventual product. Software prototyping activity is the creation of prototypes, i.e., incomplete versions of the software program being developed which may or may not be completely different from the final system we are trying to develop.

The purpose of a prototype is to allow users of the software to evaluate proposals for the design of the eventual product by actually trying them out, rather than having to interpret and evaluate the design based on descriptions. Prototyping can avoid the great expense and difficulty of changing a finished software product. This model reflects an attempt to increase the flexibility of the development process by allowing the client to interact and experiment with a working representation of the product. The developmental process only continues once the client is satisfied with the functioning of the prototype. At that stage the developer determines the specifications of the client's real needs.

Features of Prototype Model

It is a software development process that begins with requirements collection, followed by prototyping and user evaluation that facilitates to discover new or hidden requirements during the development (22). This is a cyclic version of the linear model. The goal of prototyping based development is to counter the first two limitations of the waterfall. The basic idea here is that instead of freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements. This prototype is developed based on the currently known requirements. Development of the prototype obviously undergoes design, coding and testing. But each of these phases is not done very formally or thoroughly. By using this prototype, the client can get an

"actual feel"of the system, since the interactions with prototype can enable the client to better understand the requirements of the desired system.
Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determining the requirements. In such situations letting the client "plan"with the prototype provides invaluable and intangible inputs which helps in determining the requirements for the system. It is also an effective method to demonstrate the feasibility of a certain approach. This might be needed for novel systems where it is not clear those constraints can be met or that algorithms can be developed to implement the requirements. The process model of the prototyping approach is shown in the Figure 2.5.

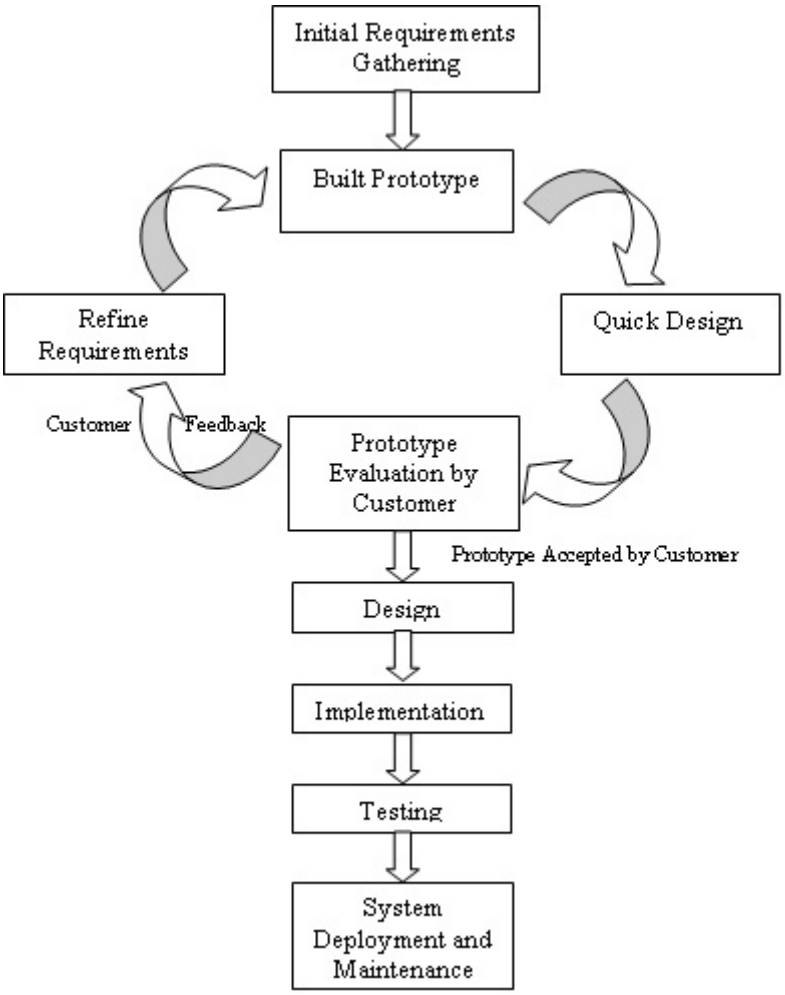


Figure 2.5: Prototype Model

The basic reason for little common use of prototyping is the cost involved in this

built-it-twice approach. However, some argue that prototyping need not be very costly and can actually reduce the overall development cost. The goal is to provide a system with overall functionality. On the other hand, the experience of developing the prototype will very useful for developers when developing the final system. This experience helps to reduce the cost of development of the final system and results in a more reliable and better designed system. Most of the successful software products have been developed using this model — as it is very difficult to comprehend all the requirements of a customer in one shot. There are many variations of this model tailored with respect to the project management styles of the companies. New versions of a software product evolve as a result of prototyping.

Advantages of Prototype Model

There are many tangable and abstract advantages of using prototyping in software development:

- **Proper clarity and 'feel'of the Proposed System :** When prototype is shown to the user, the user gets a proper clarity and 'feel'of the functionality of the software and he can suggest changes and modifications.
- **Better System Demonstration and Understanding :** Since in this methodology a working model of the system is provided, from demonstration of the prototype the users get a better understanding of the system being developed.
- **Reduces Project Risk :** It reduces project failure risks by identifying the potential risks at early and mitigation measures can be taken at time– thus effective elimination of the potential causes is possible. It also allows the software engineer some insight into the accuracy of initial project estimates and whether the deadlines and milestones proposed can be successfully met.
- **Early Error Detection :** Errors can be detected much earlier as the system is mode side by side.
- **Good and Conductive Project Environment :** Iteration between development team and client provides a very good and conductive environment during project resolve unclear objectives.
- **Better System to Users :** The client and the developers can compare if the software made matches the software specification according to which the software

program is built. It provides a better system to users as users have natural tendency to change their mind in specifying requirements and this method of developing systems supports the tendency.

- **Quicker User Feedback :** The software designer and implementer can obtain feedbacks from the users early in the project leading to better solutions.
- **Communication Interface between user and developers :** Strong Dialog between users and developers are provided through prototyping.
- **Easy Identification of Missing Functionality :** Missing functionalities can be identified easily and confusing or difficult functions can be identified as early as possible.
- **Requirements Validation :** Early involvement of user in the development process ensures requirements validation and quick implementation of incomplete, but functional application.
- **Reduced time and costs :** Prototyping can improve the quality of requirements and specifications provided to developers. Because cost of incorporating changes in the system are exponentially more to implement as they are detected later in development, the early determination user requirement really can result in faster and less expensive software.
- **Improved and increased user involvement :** Prototyping requires user involvements and allows them to see and interact with a prototype. This allow the user to provide better and more complete feedback and specifications. The presence of the prototype being examined by the user prevents many misunderstandings and miscommunication that occur when each side believe the other understands what they said. Since users know the problem domain better than anyone on the development team does, increased interaction can result in final product with better quality by satisfying the users to have desire for look, feel and performance.
- **Encourages innovative and Flexible Design :** It encourages innovation and flexible designing.

Disadvantages of Prototype Model

Using or perhaps misusing prototyping can also have following disadvantages:

- **Increased Developer Cost :** Prototyping is usually done at the cost of the developer. Sometimes the startup cost of building the development team, focused on making prototype, is high.
- **Slow Process :** It is a slow and lengthy process.
- **Frequent Requirement Change:** Requirements may change frequently that can disturb the rhythm of the development team.
- **Increased Complexity :** Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- **Insufficient analysis :** It may suffer from inadequate problem analysis. The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions, preparation of incomplete specifications or poorly engineered final projects. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable, which may not be noticed if developers are too focused on building a prototype as a model.
- **User confusion of prototype and finished system :** Too much involvement of user is not always preferred by developers. Users can begin to think that a prototype, intended to be thrown away, is the actual final system that merely needs to be finished or polished. This can lead them to expect the prototype to accurately model the performance of the final system although this is not the intent of the prototyping. Users can also become attached to features that were included in a prototype for consideration and then removed from the specification for a final system. If users are able to require all proposed features be included in the final system this can lead to feature creep.
- **Excessive development time of the prototype :** A key property to prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, they very well may try to develop a prototype that is too complex. When the prototype is thrown away the precisely developed requirements that it provides may not yield a sufficient increase in productivity to make up for the time spent developing the prototype. Users can become stuck in debates over details of the prototype, holding up the development team and delaying the final product.

- **Expectation of Higher Productivity :** A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort behind the learning curve. In addition to training for the use of a prototyping technique, there is an often overlooked need for developing corporate and project specific underlying structure to support the technology. When this underlying structure is omitted, lower productivity can often result.

- **Improper Evaluation :** Approval process and requirement is not strict thus contract may be awarded without rigorous evaluation of Prototype.

- **Difficulty in Documentation :** Identifying and documenting nonfunctional elements are difficult.

Suitability of Prototype Model

Prototyping is most beneficial in systems that will have many interactions with the users. In a scenario where there is an absence of detailed information regarding the input to the system, processing needs, output requirements– prototyping model may be employed. It has been found that prototyping is very effective in the analysis and design of on line systems, especially for transaction processing, where the use of screen dialogs is much more in evidence. This type of System Development Method is employed when it is very difficult to obtain exact requirements from the customer. Prototyping is especially good for designing good human computer interfaces. One of the most productive uses of rapid prototyping to date has been as a tool for iterative user requirements engineering and human computer interface design.
Systems with little user interaction, such as batch processing or systems that mostly do calculations benefit little from prototyping.

2.5.4 Spiral Model

History of Spiral Model

The spiral model was defined by Barry Boehm(23) in his 1988 article "A Spiral Model of Software Development and Enhancement". This model was not the first model to discuss iterative development, but was the first model to explain why the iteration matters. As originally envisioned, the iterations were typically 6 months to 2 years long. Each phase starts with a design goal and ends with the client reviewing the progress thus far. Analysis and engineering efforts are applied at each phase of the project with an eye toward the

end goal of the project. The schematic diagram of the Spiral model is represented in Figure 2.6.

Features of Spiral Model

The Spiral model combines the features of the prototyping model and the waterfall model. This process model proposes incremental development, using the waterfall model for each step with more emphasis on managing risk (23).

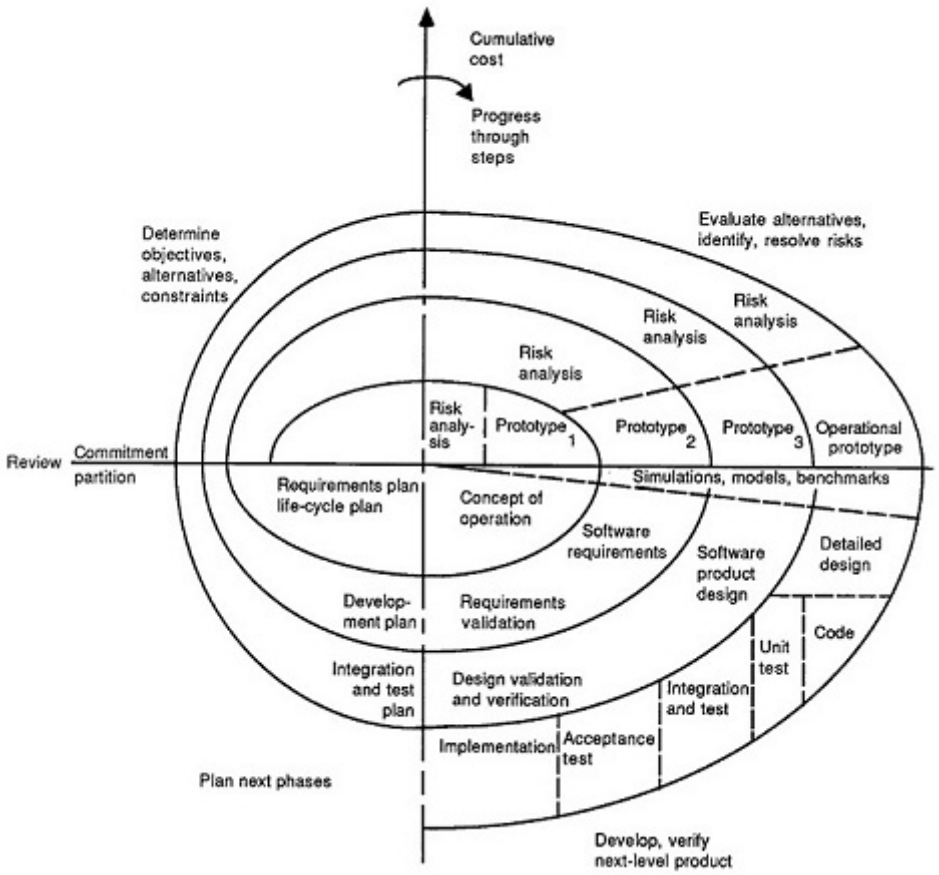


Figure 2.6: Spiral Model

The generalized steps in the spiral model are as follows :

- Step 1: The system requirements and other aspects of the systems are defined in much detail by interviewing a number of internal and external users.
- Step 2: A preliminary design is created for the new system.

- Step 3: A first prototype of the new system is constructed from the preliminary design. This is usually a scaled down system which represents an approximation of the characteristics of the final product.
- Step 4: A second prototype is evolved by a four-fold procedure:
    1. Evaluating the first prototype in terms of its strengths, weaknesses, and risks
    2. Defining the requirements of the second prototype
    3. Planning and designing the second prototype
    4. Constructing and testing the second prototype
- Step 5: Based on the customer's option, the entire project can be aborted if the risk is deemed too great. Risk factors might involve development cost overruns, operating cost miscalculation, or any other factor that could result in a less-than-satisfactory final product.
- Step 6: The existing prototype is evaluated in the same manner as was the previous prototype, if necessary another prototype is developed from it according to the fourfold procedure outlined above.
- Step 7: The preceding steps are iterated until the customer is satisfied that the refined prototype represents the final product as desired.
- Step 8: The final system is constructed, based on the refined prototype.
- Step 9: The final system is thoroughly evaluated and tested. Routine maintenance is carried out on a continuing basis to prevent large scale failures and to minimize downtime.

In spiral model, the aim of customer communication is to establish effective communication between developer and customer. The planning objectives are to define resources, project alternatives, time lines and other project related information. The purpose of the risk analysis phase is to assess both technical and management risks. The engineering task is to build one or more representations of the application. The construction and release task are to construct, test, install and provide user support by documentation and training. The purpose of customer evaluation task is to obtain customer feedback based on the evaluation of the software representation.

Advantages of Spiral Model

There are several advantages of spiral model as mentioned below:

- better project risk analysis and avoidance
- strong approval and documentation control
- implementation has priority over functionality
- additional functionalities can be added at a later date
- budget and schedule estimates become more realistic as work progresses as important issues are discovered earlier
- software engineers can get restless with protracted design processes and can start working on the project earlier
- requirements can be captured more accurately
- users see the system early
- development can be divided into smaller parts and more risky parts can be developed earlier which helps better risk management

Disadvantages of Spiral Model

Inspite of several advantages, the spiral model do have some disadvantages as below:

- Highly customized limiting re-usability
- Applied differently for each application
- Risk of not meeting budget or schedule
- Possibility to end up implemented as the waterfall framework
- Management is more complex
- End of project may not be known early
- Process is complex
- Spiral may go indefinitely
- Large numbers of intermediate stages require excessive documentation
- Doesn't work well for smaller projects

- Can be a costly model to use
- Risk analysis requires highly specific expertise

Suitability of Spiral Model

The spiral model is intended for large, expensive, and complicated mission-control projects. It is not suitable for small or low risk projects.

2.5.5 V Model

History of V Model

The V-Model in shown in Figure 2.7.

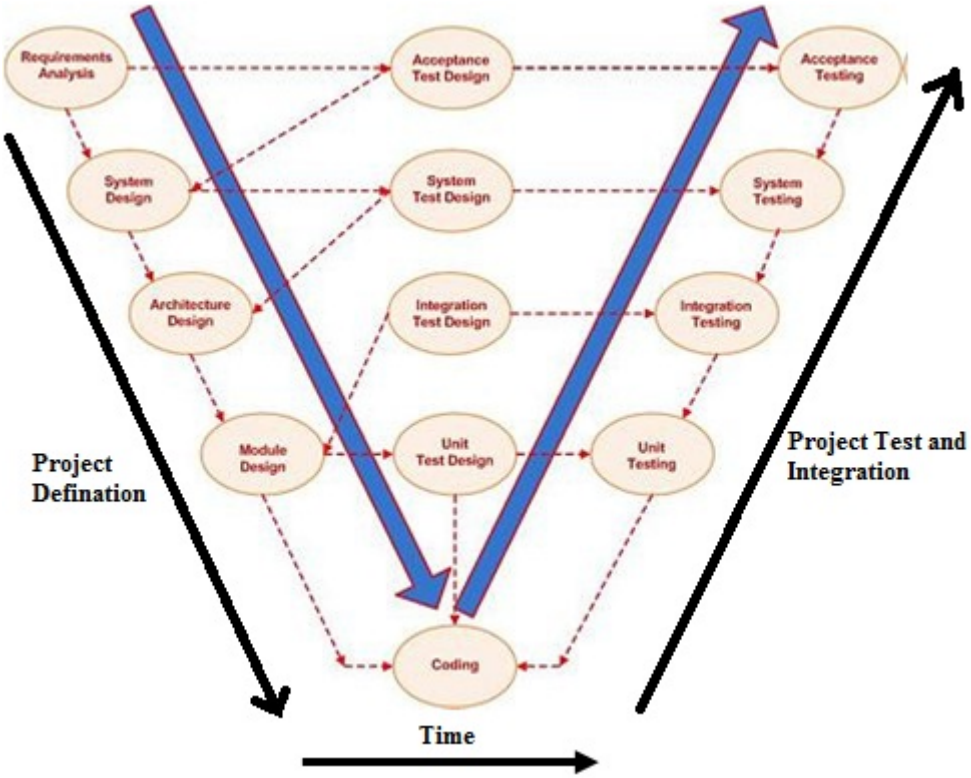


Figure 2.7: V (Shape) Model

Features of V Model

The V-model is a software development model which can be presumed to be the extension of the waterfall model which emphasizes parallelism of activities of construction and verification. Here, the process steps instead of moving down in a linear way bend upwards after the coding phase resulting in the typical V shape formation. Instead of moving down in a linear way, the process steps are bent upwards after the coding phase,

to form the typical V shape. The V-Model demonstrates the relationships between each phase of the development life cycle and its associated phase of testing.

Advantages of V Model

Following are some advantages of V model:

- simple and easy to use
- each phase has specific deliverables
- higher chance of success because of the development of test plans early on during the life cycle
- encourages definition of the requirements before designing the system and designing the software before building the components
- encourages verification and validation of all internal and external deliverables which must be testable, not just the software products
- testing activities are planned before coding which saves time and also helps in developing a very good understanding of the project at the beginning state
- it defines the products that the development process should generate

Disadvantages of V Model

The disadvantages of th V process model are:

- it is inflexible
- it is very rigid
- it has no or less ability to respond to change, if present then is difficult and expensive
- if any changes happen in mid way, then the particular documents along with requirement document has to be updated
- it produces inefficient testing methodologies
- software is developed during the implementations phase, so no early prototypes of the software are produced

- doesn't provide a clear path for problems found during testing phases
- doesn't handle concurrent event

Suitability of V Model

The V process model works well where requirements are easily understood and best suited for small projects.

2.5.6 RAD Model

History of RAD Model

Rapid Application Development (RAD) originated from rapid prototyping approaches and was first formalized by James Martin (24) during 1990s, who believed that as compared to traditional lifecycle RAD refers to a development life cycle designed for high quality systems with faster development and lower costs. By the mid 1990s the definition of RAD became used as an umbrella term to encompass a number of methods, techniques and tools by many different vendors applying their own interpretation and approach. This unstructured and extemporized ad hoc evolution of RAD means that the rationale behind its use is not always clear. It is perceived as an IS system methodology, a method for developers to change their development processes or as RAD tools to improve development capabilities (25). RAD projects are sometimes distinguished in terms of intensive and non-intensive forms. A non-intensive approach refers to projects where system development is spread over a number of months involving incremental delivery. Where as, in the intensive RAD project personnel are closeted away to achieve set objectives with a 3–6 week time frame (25). The RAD model is depicted in Figure 2.8.

RAD Model Phases

The RAD approach encompasses the following phases:

- Business Modeling:
  The information flow among business functions is defined by answering questions like:
  - what information drives the business process?
  - what information is generated?
  - who generates it?

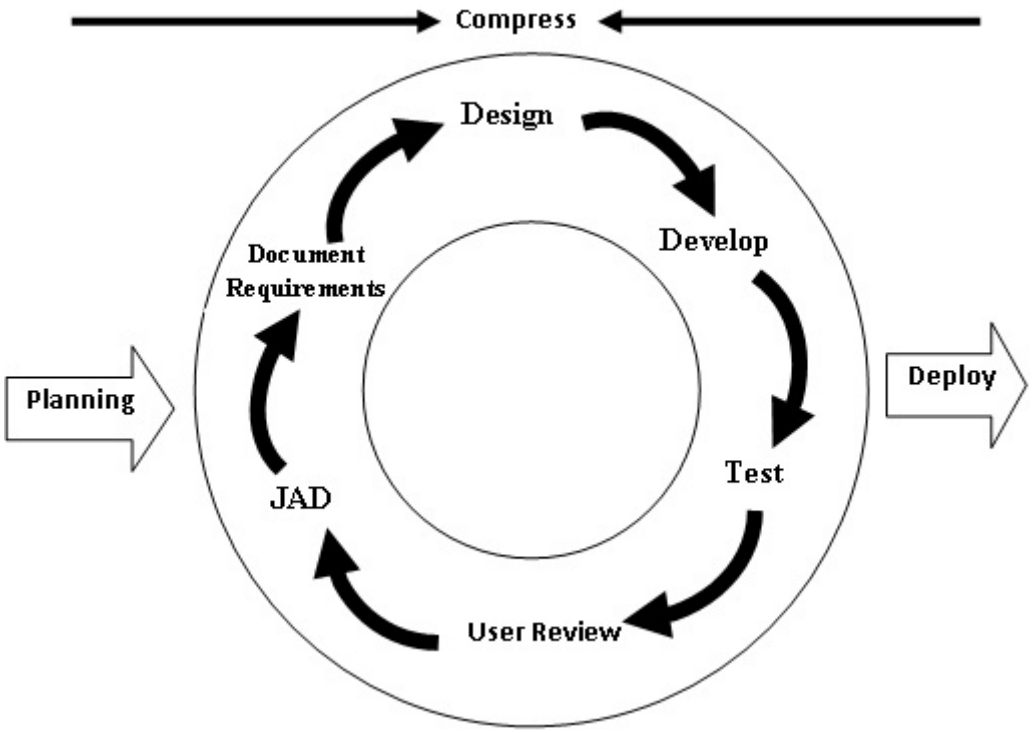


Figure 2.8: RAD Model

  - where does the information go?
  - who processes it?

  In these phase, answer to the above questions are explored and gathered.

- Data Modeling:
  The information collected from business modeling is refined into a set of data objects that are needed to support the business. The attributes are identified and the relation between these data objects is defined.

- Process Modeling:
  The data object defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting or retrieving a data object.

- Application Generation:
  Automated tools are used to facilitate construction of the software; even they use the 4th GL techniques. The RAD model assumes the use of the RAD tools like VB, VC++, Delphi etc... rather than creating software using conventional third

generation programming languages. The RAD model works to reuse existing program components or create reusable components. In all cases, automated tools are used to facilitate construction of the software.

- Testing and Turn over:
  Since the RAD process emphasizes reuse, many of the program components have already been tested. This minimizes the testing and development time. But new components must be tested and all interfaces must be fully exercised.

Features of RAD Model

Followings are some features of RAD model (24, 26, 27, 28):

- gathering requirements using workshops or focus groups
- deliver the customer's requirements with accuracy using prototyping and short iterations
- early reiterative user testing of designs
- reusing software components
- a rigid schedule that defers design improvements to the next product version
- less formality in reviews and other team communication
- timely and speedy delivery of a working application (in about 20 weeks)
- necessitates the collaboration of small and diverse teams of developers, end users and other stakeholders

The RAD DEBATE

As a systems development approach RAD has both critics and supporters whose opinions, in some cases are fundamental to individual philosophies and perceptions of its rationale. Existing literature exposes particular themes of discussion within the RAD arena and a prominent area of debate concerns the scalability of RAD across large and complex environments. Although the lack of provenance is reflected by the limited availability of published material, there is substantial reporting of its application and considerable debate about its appropriateness for different types and sizes of systems development (25, 29, 30). It is further thought that its success is linked to the

project management approach, level of management commitment, degree of end user involvement and the ability of the team to make fast authoritative decisions (31). It is also suggested that RAD projects necessitate cultural and managerial changes because people are required to behave in a different way than in the more structured traditional environments. Consequently without radical shifts in organizational attitudes and structures and peoples'mindsets many projects may fail because the change to new methodologies, methods and techniques did not fit within the culture (32). Hence the potential of a RAD development and delivery approach to meet information systems requirements in uncertain and volatile business settings of complex system development environments is questioned. Critics advocate that the need for high levels of user involvement, stakeholder collaboration, lack of project control and rigor are major issues to its success (24, 26, 27, 33).

Advantages of RAD Model

Followings are the advantages of RAD model :

- quick initial reviews are possible
- constant integration isolates problems and encourages customer feedback
- reduces the development time and reusability of components help to speedup development
- all functions are modularized so it is easy to work with

Disadvantages of RAD Model

Disadvantages of RAD model are :

- it is based on Object Oriented approach and if it is difficult to modularize the project the RAD may not work well i.e. RAD requires a system that can be modularized
- it requires highly skilled and well trained developers or engineers in the team
- both end customer and developer should be committed to complete the system in a much abbreviated time frame
- if commitment is lacking RAD will fail
- need for high levels of user involvement, stakeholder collaboration

- seeks superior project control

Suitability of RAD Model

Where the requirements cannot be locked down in the first few weeks of the project, or where requirements are not available until the later part of the project, even when the requirements push the limits of the technology, RAD is not recommended and cannot be employed. This is also true for research projects where discovery and risk play a big role and are prevalent throughout the project. Thus, RAD process works best in cases where the data is known, the requirements can be defined and kept unchanged during the development and the functional requirements can be met within a short time frame with a small team (3–4 months with 5–6 technical resources). Literature considers it more appropriate for small to medium simple, highly interactive development projects rather than for environments that are also computationally complex such as the case study involved.

2.5.7 RUP: Rational Unified Process Model

History of RUP Model

The Rational Unified Process (RUP) was brought into the IBM offering by the acquisition of the 20 year old Rational Software Corporation by IBM Software Group in February 2003. The RUP incorporates material in the areas of data engineering, business modeling, project management, and configuration management, the latter as a result of a merger with Pure-Atria in 1997. It includes elements of the Real-Time Object-Oriented Method, developed by the founders of ObjecTime, acquired by Rational in 2000 brings a tighter integration to the Rational Software suite of tools. The Rational Unified Process is the direct successor to the Rational Objectory Process (ROP), version 4, which was the result of the integration of the Rational Approach and the Objectory Process (version 3.8) after the merger of Rational Software Corporation and Objectory AB in 1995. From its Objectory ancestry, the process has inherited its process model and the central concept of use case. From its rational background, it gained the current formulation of iterative development and architecture. This ROP version 4 also incorporated material on requirements management from Requisite, Inc., and a detailed test process inherited from SQA, Inc., companies that also were acquired by Rational Software. Finally, this version of the process was the first to use the newly created Unified Modeling Language (UML 0.8). The Objectory Process was created in Sweden in 1987 by Ivar Jacobson

(34). The Rational Unified Process (RUP) has matured over the years and reflects the collective experience of the many people and companies that today make up the rich heritage of IBM's Rational Software division. Let us take a quick look at the rich ancestry of RUP 2003 (34) as illustrated in the following Figure 2.9.

The Rational Unified Process (RUP) is a software engineering process that provides

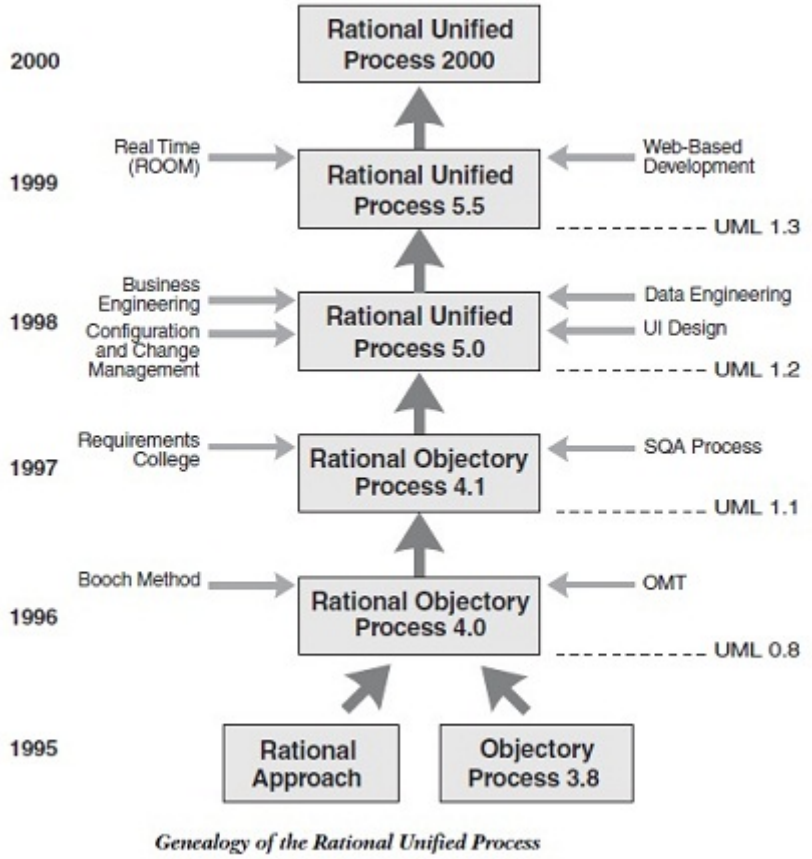


Figure 2.9: RUP History

a disciplined approach to assigning tasks and responsibilities within a development organization. The Rational Unified Process is a process product. It is developed and maintained by Rational Software and integrated with its suite of software development tools. The Rational Unified Process is also a process framework that can be adapted and extended to suit the needs of an adopting organization. The following Figure 2.10 (34) shows the structure of the RUP process:

Features of RUP Model

The Rational Unified Process captures many of the best practices in modern software development in a form that is suitable for a wide range of projects and organizations as follows :

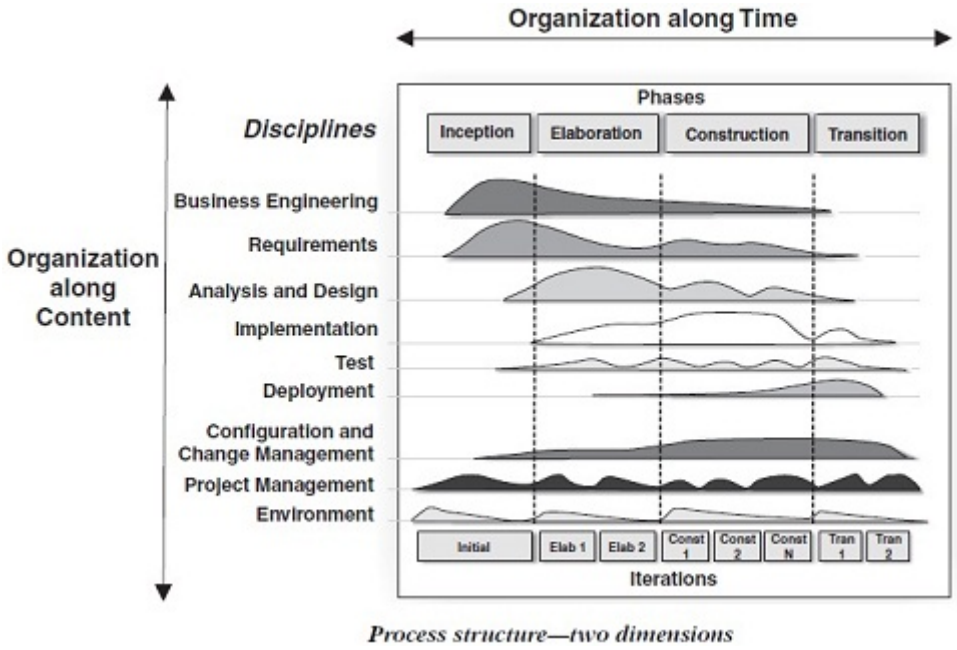


Figure 2.10: RUP (Rational Unified Process) Model

- develop software iteratively
- manage requirements
- uses component based architectures
- visually model software
- continuously verify software quality
- control changes to software

Advantages of RUP Model

The advantages of the RUP model are as follows :

- a complete methodology to manufacture software
- process with complement document facility
- openly published, distributed and supported
- supports changing requirements to meet its desired software
- supports iteration process so we can integrate the code in development life cycle in lesser time and effort spent in integration

- easy and faster code reuse resulting less development time
- there are online training and tutorials available for this process
- debugging is very easy due to component base architecture
- versatile and wide applicability
- adopts proven best industrial practices

Disadvantages of RUP Model

The disadvantages of the RUP model are as follows :

- the team members need to be expert in their field to develop the software under this methodology
- the development process is too complex and disorganized
- on cutting edge projects which utilize new technology, the reuse of components will not be possible and hence the time saving one could have made will be impossible to fulfill
- integration throughout the process of software development, in theory sounds a good thing but on particularly big projects with multiple development streams it will only add to the confusion and cause more issues during the stages of testing
- it is too complex to implement and too difficult to learn
- may lead to undisciplined form of software development

Suitability of RUP Model

More than a thousand companies were using the Rational Unified Process at the end of 2000 in various application domains, for both large and small projects. RUP is applicable in various industries like telecommunications, transportation, aerospace, defense, manufacturing, financial services, systems integrators and many more. More than 50% of these users are either using the Rational Unified Process for eBusiness or planning to do so in the near future. The RUP development team at Rational continues to refine the process for the benefit of all.

2.5.8 Evolutionary Process Model

Evolutionary development uses small, incremental product releases, frequent delivery to users, dynamic plans and processes. The evolutionary development model divides the development cycle into smaller, incremental waterfall models in which users are able to get access to the product at the end of each cycle. The users provide feedback on the product for the planning stage of the next cycle and the development team responds accordingly by changing the product, plans, process etc (35, 36). Figure 2.11 shows the structure of the EVO process(34):

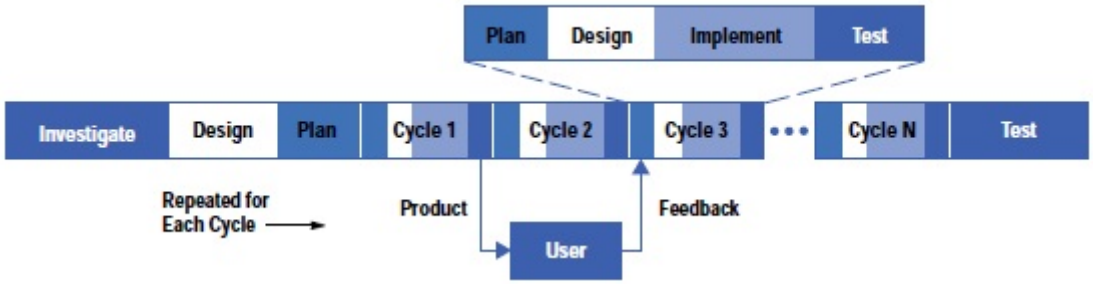


Figure 2.11: Evolutionary (EVO) development Process Model

Advantages of Evolutionary Model

Successful use of Evolutionary model can benefit not only business results but marketing and internal operations as well. The advantages of the EVO are as follows (35) :

- risk analysis is better-significant reduction in risk for software projects
- supports changing requirements
- initial operating time is less
- can reduce costs by providing a structured, disciplined avenue for experimentation
- the inevitable change in expectations when users begin using the software system is addressed by EVO's early and ongoing involvement of the user in the development process
- allows the marketing department access to early deliveries, facilitating development of documentation and demonstrations

- continuous process improvement becomes a more realistic possibility with one-to-four-week cycles
- during life cycle software is produced early which facilitates customer evaluation and feedback
- the opportunity to show their work to customers and hear customer responses tends to increase the motivation of software developers and consequently encourages a more customer focused orientation
- the cooperation and flexibility required by EVO of each developer results in greater teamwork

Disadvantages of Evolutionary Model

The disadvantages of the Evolutionary model are as follows :

- management complexity is more
- end of project may not be known which is a risk
- can be costly to use
- highly skilled resources are required for risk analysis
- project's progress is highly dependent upon the risk analysis phase

Suitability of Evolutionary Model

Evolutionary Model is better suited for large and mission-critical projects.

Chapter 3

Research Methods in Software Engineering

3.1 Introduction

Different streams of basic sciences have good explanations i.e. detailed guidance for researchers and simplified views for the stakeholders for their research strategies. While, Software Engineering (SE) is a field without too much historic background and well understood guidance since it is less than four decades old. Software engineering researchers rarely write explicitly about their paradigms of research and their standards for judging quality of results. In general, software engineering researchers seek better ways to develop and evaluate software. They are motivated by practical problems and key objectives of the research are often *quality*, *cost* and *timeliness* of software products. Many unfruitful attempts had been made to characterize software engineering research. The data suggested by Redwine-Riddle (37, 38) recommends that around 10 years of the 15–20 years of evolution are spent in concept formation, development and extension. As a result, full understanding of research strategy must account for the accumulation of evidence over times as well as for the form and content of individual projects and research papers. Software engineering will benefit from a better understanding of the research strategies that have been most successful. The model of software engineering research reflects the character of the discipline. It identifies the *types of questions* software engineers find interesting, the *types of results* we produce in answering those questions, and the *types of evidence* that we use to evaluate the results.

3.2 What Research IS and What NOT

3.2.1 What research IS?

Research is a general term which covers all kinds of studies designed to find responses to worthwhile questions by means of a systematic and scientific approach. Research could include *synthesis* and *analysis* of previous research to the extent that it leads to new and creative outcomes. Thus research is:
- *a process of investigation* i.e. a systematic inquiry that investigates hypotheses, suggests new interpretations of data or texts and poses new questions for future research to explore
- *an examination* of a subject from different points of view. It is getting to know a subject by reading up on it, reflecting, playing with the ideas, choosing the areas that interest you and following up on them
- *a hunt for the truth* i.e. a systematic inquiry to describe, explain, predict and control the observed phenomenon involving both inductive and deductive methods (39)
- *to search purposely and methodically* for new knowledge and practical solutions in the form of answers to questions formulated beforehand
- *the way you educate yourself*

Figure 3.1: Research (Analogy)

Research is defined as the creation of new knowledge and/or the use of existing knowledge in a new and creative way so as to generate new concepts, methodologies and understandings. This definition of research is consistent with a broad notion of research and experimental development as comprising of creative work undertaken on a systematic basis in order to increase the stock of knowledge, including knowledge of humanity, culture and society, and the use of this stock of knowledge to devise new applications (40).
DICTIONARY DEFINITION: RESEARCH

- *To search or investigate* exhaustively
- *Studious inquiry or examination*; especially investigation or experimentation aimed at the discovery and interpretation of facts, revision of accepted theories or laws in the light of new facts, or practical application of such new or revised theories or laws
- The *collecting of information* about a particular subject.

3.2.2 What research IS NOT?

There exist several misconception about research. In this section we justify some these misconceptions i.e. what are NOT research but often misunderstood as to be research.

- **Research Isn't Teaching:** Teaching itself is generally regarded as the synthesis and transfer of existing knowledge. Generally, the knowledge has to exist before you can teach it. Most of the time, you aren't creating new knowledge as you teach. Some lecturers may find that their students create strange new knowledge in their assignments, but making stuff up doesn't count as research either.
- **Research Isn't Scholarship:** A literature search is an important aspect of the research process but it isn't research in and of itself. The process of being a scholar generally describes surveying existing knowledge. One might be looking for new results that he/she hadn't read before, or one might be synthesizing the information for his/her teaching practice. Either way, one aren't creating new knowledge, but are reviewing what already exists.
- **Research Isn't Encyclopedic:** Encyclopedias, by and large seek to present a synthesis of existing knowledge. Collecting and publishing existing knowledge isn't research, as it doesn't create new knowledge.
- **Research Isn't Just Data Gathering:** Data gathering is a vital part of research, but it doesn't lead to new knowledge without some analysis and some further work. Just collecting the data doesn't count, unless one does something else with it.
- **Research Isn't Just About Methodology:** Just because one are using mice, or interviewing people, or using a High Performance Liquid Chromatograph (HPLC) doesn't mean one are doing research. One might be, if one are using a new data set or using the method in a new way or testing a new hypothesis. However, if one

is using the same method, on the same data, exploring the same question, then one will almost certainly get the same results, and that is repetition, not research.

- **Research Isn't Repetition, Except in Some Special Circumstances:** If one is doing the same thing that someone else has already done, then generally that isn't research unless he/she are specifically trying to prove or disprove their work. What's the difference? Repeating an experiment from 1400 isn't research. One know what the result will be before he/she start it implies it has already been verified many times before. Repeating an experiment reported earlier probably is research because the original result can't be relied upon until it is verified.

3.2.3 Science Vs. Engineering

3.2.3.1 Science and Engineering

Engineering problems deal with the creation of new artifacts and *scientific problems* deal with the study of existing ones.

Scientific research :

- is about the study phenomena and try to find the truth
- what kinds of questions are interesting?
- what kinds of results help to answer these questions, and what research methods can produce these results?
- what kinds of evidence can demonstrate the validity of a result, and how to distinguish good results from bad ones?

Engineering research :

- is about the study methods, tools, etc. that can be used to solve practical problems
- may include invention of new methods, tools, etc. or improvement of existing ones, but invention is neither necessary nor sufficient

Thus, scientific and engineering research fields can be characterized by identifying what they value:

- kinds of questions are interesting
- kinds of results to answer the interested questions

- kind of research methods that can produce these results
- kinds of evidence can demonstrate the validity of a result
- way of distinguishing good and bad results

3.2.3.2 Difference between the Objectives of Science and Engineering Study

Science pays attention to natural aspects while engineering is concerned with artificial aspects. While science deals with the study of what things are like, engineering is concerned with what they should be like in order to make it possible to construct new objects. While sciences deal with the study of existing objects and phenomena, be it physically, metaphysically or conceptually, but engineering is based on how to do things, how to create new objects. The difference between science and engineering lies in the modes of knowledge and action that they develop, not in one of them knowing and the other applying (41). Nevertheless, both science and engineering are knowledge and action since in the same way science is also action, not just knowledge; correlatively, we may say that engineering is also knowledge and not just application. In other words, science and engineering will differ in the research process which is used in each one (42). An engineering problem becomes a scientific problem once the object has been created; when a new artifact is created for example a new model, by an engineering research process, this new artifact becomes an existing one being an object of study by a scientific research process for example, studying its correctness, its quality, etc.

3.2.3.3 Software Engineering : Science or Engineering?

There are several non-conclusive opinions by several authors (43, 44, 45) regarding the issues whether software engineering can be a real engineering. To answer this question, we take as reference point other fields with a bigger historic background and also with more maturity such as Electronics, Chemistry or Geology– in such fields may we talk as science as engineering. So, in the same way electronic physics and electronic engineering, chemistry and chemical engineering, geology and geological engineering coexist and making a simile with these fields, we can say that SE has a double nature of science and engineering, depending on the object of study; this fact determines the research process. The nature of Software Engineering (SE) research, basically science or engineering, depends on its object of study. Seen this way, and according to the object of study, the research process will be different so that the kinds of problems can be tackled by means

of different research methods and even by means of different paradigms. A classification of the issues of the SE discipline in:

- *those with a scientific nature :* focuses on theoretical bases of the SE
- *those with an engineering nature :* deals with the problem of building new software artifacts.

We try to justify this hypothesis on the basis of the paradigms and the research process which is in general used for the resolution of these kinds of problems.
Therefore, using Blum terminology (46), software science comes from computer science and pays attention to the aspects that have to do with the study of built artifacts, like code or other kind of artifacts such as models, documents, etc. This science deals with problems such as algorithmic complexity, software metrics, testing techniques, etc. On the other hand, we can find other kind of engineering research problems, supported by software science, concerned with the creation of software artifacts and we could define it as the study of transforming ideas into operations (46).
Although there are other classifications of problems in SE, these classifications are not appropriate since they are not focused on the research process. Thus, for instance, the SE Body of Knowledge project (SWEBOK) (47) sets out areas as: Software design, Software construction, SE tools and methods, etc. This schema is accurate enough but it is mainly centered on the creation of a body of knowledge with educative aims. Then, it provides invalid areas of knowledge for a classification of research problems because we can find scientific and engineering problems in each area, for example in SE tools and methods area. It is not the same trying to create a new method than testing a previously created method. In Software design, you can either create new models to improve the design process or study existing models, and analyze their implantation and use in a company. We notice the former case is an engineering problem while the latter case is an empirical or socio-cultural problem but all of them are included in the same area of the SWEBOK. This inclusion into the same area makes its difference in the research field impossible.
Hence, it is well justified that software engineering is an engineering discipline.

3.3 Broader View of Software Engineering Research Paradigm

To carry out research in any field, it is important to set out a paradigm for any research to realize.

3.3.1 Types of Research Paradigms

In general, we may divide the research paradigm into the following two categories (48):

- **Descriptive Paradigms:** The descriptive paradigms is evaluative–deductive or positivist paradigm, valuative interpretive or interpretive paradigm, evaluative-critical or critical paradigm etc.
- **Formulative Paradigms:** The formulative paradigms are formulative model, formulative process, method, algorithm etc.

3.3.2 Types of Research Problem Domain

From our point of view, depending on the nature of problems, we may broadly classify the problem domains in to the following two types:

- **Scientific Problems Domain :** In scientific research problems, *evaluative* paradigms are the used in most cases. *Positivist paradigms* as used in empirical sciences and either *interpretive* or *constructive* paradigms as used in social and cultural problems. In this regard, for instance, we could apply positivist paradigms to testing or interpretive paradigms to organization processes which are necessary for the implantation of a tool. Although there are other paradigms, even combinations of paradigms which can give rise to mixed paradigms, they always present characteristics which allow us to include them in a behavioral science research paradigm (49).
- **Engineering Problems Domain :** To engineering research problems, *descriptive paradigms* are used and these paradigms interact with positivist and interpretive paradigms (50). Thus, for instance, by means of literature reviews the researcher can try to establish the weak spots of a model and its respective technique of creation and afterwards can try to establish a description of a new technique and the new built model.

Brooks (51) reflected on the tension in human computer interaction research between :

- "narrow truths" proved convincingly by statistically sound experiments that satisfy the gold standard of science, and
- "broad truths", generally applicable, but supported only by possibly unrepresentative observations that provide pragmatic guidance, but at risk of over generalization.

Brooks (51) proposes a certainty shell structure – to recognize *three* nested classes of results :

- **Findings:** well established scientific truths, judged by truthfulness and rigor
- **Observations:** reports on actual phenomena, judged by interestingness
- **Rules of thumb:** generalizations, signed by their author but perhaps incompletely supported by data, judged by usefulness with freshness as a criterion for all three.

This same problem is also in software engineering. Observations and rules of thumb provide valuable guidance for practice when findings are not available. They also help to understand the area and lay the groundwork for the research that will yield findings in due time.
Further, Newman (52) compared research in human computer interaction (HCI) to research in engineering. He characterized engineering practice, identified three main types of research contributions, and performed a preliminary survey of publications in five engineering fields. He found that over 90% of the contributions was of three kinds:

- EM-Enhanced analytical modeling techniques, based on relevant theory, that can be used to tell whether the design is practicable or to make performance predictions
- ES-Enhanced solutions that overcome otherwise insoluble aspects of problems or that are easier to analyze with existing modeling techniques
- ET-Enhanced tools and methods for applying analytical models and for building functional models or prototypes.

3.4 Prior Reflections on Software Engineering Research

In the 1950s there was certain research but it was covered of confusion and without any significant publication. This idea can be supported by means of the fact that its first publications and conferences were held in the late 1960s (53). Till the beginning of 1980s, the existence of computer science and software engineering were cohesive– almost non-distinguishable.

From early 1980s onwards the academic presence of software engineering begins to separate from computer science. This influenced a lot in SE research developing (48). This youth of the discipline of SE is resulted in an immaturity of this research field. This immaturity is verified because research which is carried out in this discipline has several deficiencies (54, 55, 56) like the lack of systematic rigorous method, the lack of the evident methods of validation, etc. We can say SE research still lacks suitable scientific precision.

Each scientific discipline has a certain object of study which distinguishes the process to be followed in the research. The problem of SE research basically takes root in the fact that it is not so evident which the object of study is or rather the problem is that there are several objects of study with different nature and so, there are also different research and validation processes. This problem is motivating a rising concern with research methods and the validation in SE (41, 48, 50, 57, 58).

In 1980, Mary Shaw (43) examined the relation of engineering disciplines to their underlying craft and technology and laid out expectations for an engineering discipline for software. In 1984–85, Redwine, Riddle, and others (37, 38) proposed a model for the way software engineering technology evolves from research ideas to widespread practice. More recently, software engineering researchers have criticized common practice in the field for failing to collect, analyze and report experimental measurements in research reports (54, 55, 56, 59). Redwine and Riddle (37) presented timelines for several software technologies as they progressed through these phases up until the mid 1980s. Mary Shaw (60) presented a similar analysis for the maturation of software architecture in the 1990s. In 2001, Mary Shaw (60) presented preliminary sketches of some of the successful paradigms for software engineering research drawing heavily on examples from software architecture. Redwine and Riddle (37, 38) reviewed a number of software technologies to see how they develop and propagate and found that it typically takes 15–20 years for a technology to evolve from concept formulation to the point where it's ready for popularization.

Software engineering research is targeted to improve the practice of software development, so research planning should make provisions for the transition. The IMPACT project (61) is tracing the path from research into practice. The objectives of the project include identifying the kinds of contributions that have substantial impact and the types of research that are successful.

3.5 Types of Software Engineering Research

3.5.1 Types of General Research

In general, research may be categorized in to different types as mentioned below:

- **Action Research :** Action research is a methodology that combines action and research to examine specific questions, issues or phenomena through observation and reflection, and deliberate intervention to improve practice.
- **Applied Research :** Applied research is research undertaken to solve practical problems rather than to acquire knowledge for knowledge sake.
- **Basic Research :** Basic research is experimental and theoretical work undertaken to acquire new knowledge without looking for long term benefits other than the advancement of knowledge.
- **Clinical Trials :** Clinical trials are research studies undertaken to determine better ways to prevent, screen for, diagnose or treat diseases.
- **Epidemiological Research :** Epidemiological research is concerned with the description of health and welfare in populations through the collection of data related to health and the frequency, distribution and determinants of disease in populations, with the aim of improving health.
- **Evaluation Research :** Evaluation research is research conducted to measure the effectiveness or performance of a program, concept or campaign in achieving its objectives.
- **Literature Review :** Literature review is a critical examination, summarization, interpretation or evaluation of existing literature in order to establish current knowledge on a subject.

- **Qualitative Review :** Qualitative research is research undertaken to gain insights concerning attitudes, beliefs, motivations and behaviors of individuals to explore a social or human problem and include methods such as focus groups, in-depth interviews, observation research and case studies.
- **Quantitative Review :** Quantitative research is research concerned with the measurement of attitudes, behaviors and perceptions and includes interviewing methods such as telephone, intercept, *door-to-door* interviews as well as self-completion methods such as mail outs and online surveys.
- **Service or Program Monitoring and Evaluation :** Service or program monitoring and evaluation involves collecting and analyzing a range of processes and outcome data in order to assess the performance of a service or program, and to determine if the intended or expected results have been achieved.

Typically our research falls under the category of applied research where we undertook to solve some issues related to software crisis following a typical process model proposed by us – named BRIDGE (62).

3.5.2 Types of Software Engineering Research

Software engineering (SE) research maybe one of the following types:

1. Method or means of development
2. Method for analysis or evaluation
3. Design, evaluation, or analysis of a particular instance
4. Generalization or characterization
5. Feasibility study or exploration

The *first two types of SE research* i.e. method or means of development and Method for analysis or evaluation, produce methods of development or of analysis that the authors investigated in one setting, but that can presumably be applied in other settings.
The *third type of SE research* i.e. Design, evaluation, or analysis of a particular instance, deals explicitly with some particular system, practice, design or other instance of a system or method; these may range from narratives about industrial practice to analytic comparisons of alternative designs. For this type of research the instance itself should

have some broad appeal — an evaluation of Java is more likely to be accepted than a simple evaluation of the toy language you developed last summer.
The *fourth type of SE research* i.e. generalizations or characterizations explicitly rise above the examples presented in the paper.
Finally, the *fifth type of SE research* i.e. feasibility study or exploration types of research deal with an issue in a completely new way are sometimes treated differently from papers that improve on prior art.
The most common kind software engineering research reports an improved method or means of developing software—-that is, of designing, implementing, evolving, maintaining, or otherwise operating on the software system itself. Also fairly common kinds of software engineering research are about methods for reasoning about software systems, principally analysis of correctness i.e. testing, verification and validation (63).

This research work, in particular, from the software engineering perspective falls under the first category i.e. method or means of development. We proposed a software development method of in the form of a typical process model - named BRIDGE (62).

3.6 Typical Phases of Research

Redwine and Riddle (37, 38) identify *six* typical phases of research:

- **Phase 1: Basic Research :** Investigate basic ideas and concepts, put initial structure on the problem, and frame critical research questions.
- **Phase 2: Concept Formulation :** Circulate ideas informally, develop a research community, converge on a compatible set of ideas, and publish solutions to specific sub-problems.
- **Phase 3: Development and Extension :** Make preliminary use of the technology, clarify underlying ideas, and generalize the approach.
- **Phase 4: Internal Enhancement and Exploration :** Extend approach to another domain, use technology for real problems, stabilize technologies, develop training materials, show value in results.
- **Phase 5: External Enhancement and Exploration :** Similar to internal, but involving a broader community of people who weren't developers, show substantial evidence of value and applicability.

- **Phase 6: Popularization :** Develop production quality, supported versions of the technology, commercialize and market technologies and expand user community.

3.7 Research Strategies

3.7.1 Creating Research Strategies

The spectrum of good research strategies includes experimental computer science (54, 55, 56, 59) and this spectrum is much broader than just experimental research. Of course, not all the combinations of question, result and validation make sense, but often many of such combinations do.

3.7.2 Building Good Research Results

This discussion on research results has focused on individual results as reported in conference and journal papers. Major results, however, gain credibility over time as successive papers provide incremental improvement of the result and progressively stronger credibility. Assessing the significance of software engineering results should be done in this larger context. As increments of progress appear, they offer assurance that continued investment in research will pay off. Thus initial reports in an area may be informal and qualitative, but it presents a persuasive case for exploratory research. But reports afterwards in this area present empirical case, and later formal models that justify larger investment. This pattern of growth is consistent with the Redwine-Riddle's (37, 38) model of technology maturation.

3.8 Classifications of Research Design Strategies in Software Engineering

Research design strategies in software engineering may be classified into the following categories:

- Empirical
- Observational
- Correlational
- Quasi Experimental

The above research design strategies are discussed in the following section in brief.

3.8.1 Empirical research methods in software engineering

3.8.1.1 Controlled Experiments – high level of control and repeatable

A true experiment is defined as an experiment conducted where an effort is made to impose control over all other variables except the one under study. In controlled experiments, researchers control independent variable(s) and observe dependent variables(s). It is often easier to impose this sort of control in a laboratory setting. Thus, true experiments have often been erroneously identified as laboratory studies. This type of research design are useful for studying isolated activities in controlled environments i.e. comparing the use of two programming languages using students as subjects. These types of research are also called as *research in-the-small*.

3.8.1.2 Surveys – Sampling and Statistically Valid

In survey method research, participants answer questions administered through interviews or questionnaires. After participants answer the questions, researchers describe the responses given. In order for the survey to be both reliable and valid it is important that the questions are constructed properly. Another consideration when designing questions is whether to include open-ended, closed-ended, partially open-ended, or rating-scale questions (64). *Open-ended questions* allow for a greater variety of responses from participants but are difficult to analyze statistically because the data must be coded or reduced in some manner. *Closed-ended* questions are easy to analyze statistically, but they seriously limit the responses that participants can give. Many researchers prefer to use a *partiall open-ended or Likert-type* scale because it's very easy to analyze statistically. In this method, researchers observe a phenomenon in a representative subset of some population i.e. collecting data about a number of projects in different organization through questionnaires and/or interviews. This types of research is also called *research in-the-large* (61).

3.8.1.3 Case Studies

Case study research involves an in-depth study of an individual or group of individuals. Case studies often lead to testable hypotheses and allow us to study rare phenomena. Case studies should not be used to determine cause and effect, and they have limited use for making accurate predictions. Case studies needs careful documentation, multiple case studies can strengthen results. But, there are two serious problems with case studies

i.e. expectancy effects and atypical individuals. *Expectancy effects* include the experimenters'underlying biases that might affect the actions taken while conducting research. These biases can lead to misrepresenting participants descriptions. Describing *atypical individuals* may lead to poor generalizations and detract from external validity. In case study, researchers observe phenomena in a *real-life context* i.e. collecting data about an ongoing project. Possible methods for data collection include interviews, project participation, documents, artifacts (software). This type of research is also called *research in-the-typical* (61).

3.8.2 Observational Method/Field Observation

The primary characteristic of each of observational method (61) is that phenomena are being observed and recorded. A detailed report with analysis would be written and reported constituting the study of this individual case. These studies may also be qualitative in nature or include qualitative components in the research. There are two main categories of observations i.e. Quantitative and Qualitative. *Quantitative observational method* is generally based on measurement. While, *qualitative observational method* is generally based on no measurement.

There are two main categories of the observational method i.e. naturalistic observation and laboratory observation. The biggest advantage of the naturalistic method of research is that researchers view participants in their natural environments. Proponents of laboratory observation often suggest that due to more control in the laboratory, the results found when using laboratory observation are more meaningful than those obtained with naturalistic observation. Laboratory observations are usually less time consuming and cheaper than naturalistic observations. Of course, both naturalistic and laboratory observation are important in regard to the advancement of scientific knowledge.

When the science has an empirical nature, quantitative research methods can be applied (65); these methods try to solve problems like: *what model method is more efficient?* But, when the science has a social and cultural nature, qualitative research methods can be applied (66) and these methods can seek to answer questions like: *what factors make a given software process unacceptable to the company?* or *why is one information systems development tool more acceptable than another?*

3.8.3 Correlational Research

In general, correlational research examines the co-variation of two or more variables. Correlational research can be accomplished by a variety of techniques which include the collection of empirical data. Often, correlational research is considered type of observational research as nothing is manipulated by the experimenter or individual conducting the research. It is important to note that correlational research is not causal research. In other words, we cannot make statements concerning cause and effect on the basis of this type of research. Correlational research is often conducted as exploratory or beginning research. Once variables have been identified and defined, experiments are conductible.

3.8.4 Quasi-Experimental

Quasi-experiments are very similar to true experiments but use naturally formed or pre-existing groups. Therefore, this cannot be a true experiment. When one has naturally formed groups, the variable under study is a subject variable as opposed to an independent variable. As such, it also limits the conclusions we can draw from such a research study. There are many differences between the groups that we cannot control and those could account for differences in our dependent measures. Thus, we must be careful concerning making statement of causality with quasi-experimental designs. However, there are also instances when a researcher designs a study as a traditional experiment only to discover that random assignment to groups is restricted by outside factors. The researcher is forced to divide groups according to some pre-existing criteria. The results are again restricted due to the quasi-correlational nature of the study. As the study has pre-existing groups, there may be other differences between those groups than just the presence or absence of a wellness program. Hence, quasi-experiments may result from either studying naturally formed groups or use of pre-existing groups. When the study includes naturally formed groups, the variable under study is a subject variable. When a study uses pre-existing groups that are not naturally formed, the variable that is manipulated between the two groups is an independent variable. As no random assignment exists in a quasi-experiment, no causal statements can be made based on the results of the study.

3.9 Software Engineering Research Model: Questions, Results and Validation

Over several years a model was evolved, that explains software engineering research papers by classifying:

- the types of research questions they ask
- the types of results they produce, and
- the character of the validation they provide

3.9.1 Software Engineering Research Questions and Types

Generally speaking, software engineering researchers seek better ways to develop and evaluate software. *Development* includes all the synthetic activities that involve creating and modifying the software, including the code, design documents, documentation, etc. *Evaluation* includes all the analytic activities associated with predicting, determining, and estimating properties of the software systems including both functionality and extra-functional properties such as performance or reliability.
Software engineering research answers questions (63):
a. About method or means of development
- how can we do/create (or automate doing) X?
- what is a better way to do/create X?

b. About method for analysis
- how can I evaluate the quality/correctness of X?
- how do I choose between X and Y?

c. About details of design, evaluation, or analysis of a particular instance
- what is a (better) design or implementation for application X?
- what is property X of artifact/method Y?
- how does X compare to Y?
- What is the current state of X / practice of Y?

d. About Generalization or characterization over whole class of systems or techniques
- given X, what will Y (necessarily) be?
- what, exactly, do we mean by X?
- what are the important characteristics of X?

- what is a good formal/empirical model for X?
- what are the varieties of X, how are they related?

e. About exploratory issues concerning existence or Feasibility

- feasibility Does X even exist, and if so what is it like?
- is it possible to accomplish X at all?

In this work, we answer question about method or means of development.

3.9.2 Software Engineering Research Results and Types

The tangible contributions of software engineering research may be procedures or techniques for development or analysis; they may be models that generalize from specific examples, or they may be specific tools, solutions, or results about particular systems. Software engineering research results may take one of the following form:

a. Procedure or Technique

- New or better way to do some task, such as design, implementation, measurement, evaluation, selection from alternatives,
- Techniques for implementation, representation, management, and analysis, but not advice or guidelines.

b. Qualitative or Descriptive model

- Structure or taxonomy for a problem area; architectural style, framework, or design pattern; informal domain analysis
- Well-grounded checklists, well-argued informal generalizations, guidance for integrating other results.

c. Empirical model

- Empirical predictive model based on observed data

d. Analytic model

- Structural model precise enough to support formal analysis or automatic manipulation

e. Notation or tool

- Formal language to support technique or model (should have a calculus, semantics, or other basis for computing or inference)
- Implemented tool that embodies a technique

f. Specific solution

- Solution to application problem that shows use of software engineering principles – may be design, rather than implementation
- Careful analysis of a system or its development

- Running system that embodies a result; it may be the carrier of the result, or its implementation may illustrate a principle that can be applied elsewhere

g. Answer or judgment

- Result of a specific analysis, evaluation, or comparison

h. Report

- Interesting observations, rules of thumb

The result may be a specific procedure or technique for software development or for analysis. It may be more general, capturing a number of specific results in a model; such models are of many degrees of precision and formality. Sometimes, the result is the solution to a specific problem or the outcome of a specific analysis. Finally, as Brooks observed (51), observations and rules of thumb may be good preliminary results.

By far the most common kind of software engineering research result reports a new procedure or technique for development or analysis either described in narration or embodied in a tool. Analytic and descriptive models are also common. The analytic models support predictive analysis, whereas descriptive models explain the structure of a problem area or expose important design decisions. Models of various degrees of precision and formality are also common, with better success rates for quantitative than for qualitative models. Tools and notations were well represented, usually as auxiliary results in combination with a procedure or technique. But, empirical models backed up by good statistics are uncommon.

Our research result took the form of a descriptive model.

3.9.3 Research Result Validation Techniques

A key concept relevant to a discussion of research methodology is that of validity. The validity is the question about the truthfulness of a study under consideration. There are four types of validity that can be discussed in relation to research and statistics. Thus, when discussing the validity of a study, one must be specific as to which type of validity is under discussion. Good research requires not only a result, but also clear and convincing evidence that the result is sound. This evidence should be based on experience or systematic analysis, not simply persuasive argument or textbook examples.

3.9.3.1 Foundations for Acceptance of Research Results

The acceptance of any research results depends heavily on:

- the process of obtaining the results

- analysis of the results themselves

Research yields new knowledge. This knowledge is expressed in the form of a particular result.

3.9.3.2 Types of Validity of Empirical Studies

There are four types of validity. Each type of validity has many threats which can pose a problem in a research study. For a comprehensive discussion of the four types of validity, the threats associated with each type of validity, and additional validity issues one may go through Cook and Campbell (67). Each of the four types of validity will be briefly defined and described below.

a. Statistical Conclusion Validity

According to Cook and Campbell (67), "statistical conclusion validity refers to inferences about whether it is reasonable to presume covariation given a specified alpha level and the obtained variances". Essentially, the question that is being asked is:
- are the variables under study related? or
- is variable A correlated (does it co-vary) with Variable B?
If a study has good statistical conclusion validity, we should be relatively certain that the answer to these questions is "yes". Examples of issues or problems that would threaten statistical conclusion validity would be random heterogeneity of the research subjects and small sample size.

b. Construct Validity

One is examining the issue of construct validity when one is asking the questions:
- am I really measuring the construct that I want to study? or
- Is my study confounded (Am I confusing constructs)?
Threats to construct validity include subject apprehension about being evaluated, hypothesis guessing on the part of subjects and bias introduced in a study by expectancies on the part of the experimenter.

c. Internal Validity

Once it has been determined that the two variables (A & B) are related, the next issue to be determined is one of causality. One is examining the issue of internal validity when one is asking the questions:
- does A cause B?
If a study is lacking internal validity, one cannot make cause and effect statements based

on the research; the study would be descriptive but not causal. There are many potential threats to internal validity.

d. External Validity

External validity addresses the issue of being able to generalize the results of your study to other times, places, and persons. Therefore, one needs to ask the following questions to determine if a threat to the external validity exists:
- would I find these same results with a difference sample?,
- would I get these same results if I conducted my study in a different setting?, and
- would I get these same results if I had conducted this study in the past or if I redo this study in the future?
If I cannot answer "yes" to each of these questions, then the external validity of my study is threatened.

3.9.3.3 Software Engineering Research Validation Techniques

The most common kinds of validation are experience in actual use and systematic analysis. The other result validation methods applied in software engineering based on assertion, demonstration, examples, evaluation, persuasion, or offer no evidence at all.

- **Experience**
  My result has been used on real examples by someone other than me, and the evidence of its correctness / usefulness / effectiveness is:
  - ... narrative
  - ... data, usually statistical, on practice
  - ... comparison of this with similar results in actual use
- **Analysis**
  I have analyzed my result and find it satisfactory through:
  - ... rigorous derivation and proof
  - ... data on controlled use
  - ... experiment
- **Example**
  Here's an example of how it works on:

  - ... a toy example, perhaps motivated by reality
  - ... a system that I have been developing

- **Persuasion**
  If the original question was about feasibility, a working system, even without analysis, can be persuasive:
  - ... if you do it the following way ...
  - ... a system constructed like this would ...
  - ... this model seems reasonable

- **Evaluation**
  Given the stated criteria, my result:
  - ... adequately describes the phenomena of interest
  - ... accounts for the phenomena of interest
  - ... is able to predict ... because ..., or ... gives results that fit real data ...
  - Feasibility studies, pilot projects

- **Blatant assertion**
  In case of blatant assertion generally no serious attempt to evaluate result is taken.

3.10 Software Engineering Research: The Road-map

3.10.1 Software Engineering Research Methods

Software engineering research includes, but is not limited to, experimental research. Depending on the kind of problem to solve and the context of the problem, science or engineering, different research methods are used (68). Moreover, scientific research methods cannot always be applied to engineering research problems (69). Scientific research problems are similar to problems broached in traditional sciences and can have either an empirical or a cultural and social nature.
When the science has an empirical nature, quantitative research methods can be applied (65); and when the science has a social and cultural nature, qualitative research methods can be applied (66). In both, it is necessary certain knowledge of the reality: the object of study is an existing object in the world. Thus, this kind of problems use the research methods proposed by traditional sciences, as they study phenomena and objects of the

world regardless of how they were created. However, there is not any precise method to broach engineering research problems and the search for a method appropriate to this field is becoming a research field in its own right (48, 50, 57, 58, 66). The solution of problems purely concerning engineering requires methods of a different kind since in these cases it is directly possible to apply neither empirical methods nor methods which have to do with social and cultural component as the object of study does not yet exist (70). Furthermore, in the case of engineering, it is necessary a major component of creativity, which makes it difficult to draw up a universal method for solving problems within this field. For instance, *what research method would be valid for the specification of a new methodology for software development?* It would be necessary to study existing methodologies, reflecting on them to determine their advantages and disadvantages and proposing a new one, which, while retaining the advantages of the methodologies studied, would, as far as possible, lack their shortcomings. Arriving at a better final proposition would largely depend on the creativity and common sense applied to the construction of the new method. This method is applied in engineering consist in the formulation of experiences and the identification of the best practices (45). In 1993, Basili laid out experimental research paradigms appropriate for software engineering (71).

3.10.2 Critiques of Experimental Software Engineering

During 1995–88, Later, Tichy (54, 55) and colleagues criticized the lack of quantitative experimental validation reported in conference papers:
"Computer scientists publish relatively few papers with experimentally validated results... The low ratio of validated results appears to be a serious weakness in CS research... This weakness should be rectified (54)".
They classified 246 papers in computer science and, for comparison, 147 papers in two other disciplines, according to the type of contribution in the article. The majority of the papers produced design and modeling results. Then, they assess each papers'evaluation of its results on the basis of the fraction of the article's text devoted to evaluation. They found, that hypothesis testing was rare in all samples, that a large fraction (43%) of computer science design and modeling papers lacked any experimental evaluation, and that software engineering samples were worse than computer science in general (72).
Zelkowitz and Wallace (56, 59) built on Basili's description of experimental paradigms and evaluated over 600 computer science papers and over 100 papers from other disciplines published over a 10-year period. Again, they found that too many papers have no experimental validation or only informal validation, though they did notice some progress

over the 10-year period covered by their study. These critiques start from the premise that software engineering research should follow a classical experimental paradigm.

Chapter 4

Emergence of Component Based Software Engineering

4.1 Introduction

In early days, software engineering approach was ad hoc. Around 1970s, introduction of structured programming" gave a formal shift in software engineering from the ad hoc to a systematic approach. Then around 1980s, introduction of object oriented programming with some advancement explores new areas in software engineering. In recent dates, with the introduction of Component Based Software Development (CBSD), the industry is moving in a new direction. The basic insight is that most software systems are not new. Rather, they are variants of systems that have already been built. This insight can be leveraged to improve the quality and productivity of the software production process (73). These days software systems are more complex as compared to those of early. These complex, high quality software systems are built efficiently using component based approach in a shorter time. Component based systems are easier to assemble and therefore less costly to build than developing such systems from scratch. The importance of component based development lies in its efficiency. In addition, CBSE encourages the use of predictable architectural patterns and standard software infrastructure, thereby leading to a higher result. In the remaining part of this thesis the term "component"and "software components"will be used interchangeably.

Based on the publication in the "*International Journal of Advanced Research in Computer Science and Software Engineering(IJARCSSE)*", ISSN: 2277 128X, 2(3), 311–315, 2012.

4.2 The journey of the Component Based Software Development (CBSD) Era

In 1968, Douglas McIlroy's (74) first share the idea of Component Based Software Development (CBSD) at the NATO conference on software engineering in Garmisch, Germany in his paper titled "Mass Produced Software Components". He discussed the idea that, software may be componentized i.e. built from pre-developed software components. His subsequent inclusion of pipes and filters into the Unix operating system was the first implementation of an infrastructure for this idea. Later, Brad Cox set out to create an infrastructure and market for these components by inventing the Objective-C programming language. IBM led the path with their System Object Model (SOM) in the early 1990s. Some claim that Microsoft paved the way for actual deployment of component software with OLE and COM. As of 2010 many successful software component models do exist.

4.3 Understanding Software Components

In early days, the principles of software engineering have been focused in developing software system from the very scratch development for individual software. It means that, for all the functionalities to be supported by a system have been designed and coded individually for the proposed systems under development. Brown (75) posits that it would be infeasible for developers and organizations to consider constructing each new information system from scratch. Instead, information systems would need to be developed with reused practices, software components and products that have been tested and proven to be effective and efficient in order to remain in business and gain competitive advantage (75, 76, 77). Upon long observation it was found that- there are certain functionalities those are common in many systems. Hence, if these common functionalities can be developed independently - may be reused in different systems without redevelopment from scratch. Later, these can be integrated to any system as part whenever it is suitable! At the same time, it will reduce the development effort of such systems as there is no need to develop the same common parts again and again for different systems. This originates the concepts of Software Components. Although, a software components is a small parts of a system, but often a large system as a whole may be seen as a software component as well. It is also possible that, the system consists of components is a component itself. In all cases, the components are required to be reusable components after all.

Historically, "component"in software is a rough synonym for "module"or "unit"or "routine". A generally accepted view of a software component is that, it is *a software unit with provided services and required services from others too* (Figure 4.1). The provided services are operations performed by the component whereas, the required services are the services needed by the component to provide target services. The one or more interface of a component consists of the specifications of its provided and required services (78).

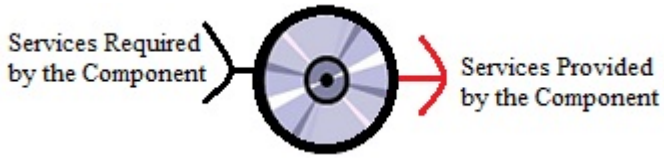


Figure 4.1: Software Components (An Abstract View)

As component is simply a data capsule, information hiding becomes the core construction principle underlying components. Clemens Szyperski (79) suggests shifting the focus away from code source. He defines a software component as executable, with a black-box interface that allows it to be deployed by those who did not develop it. It is important for a software component to be easily combined and composed with other software components. This is because a software component will only achieve its usefulness when it is used in collaboration with other software components (80).
According to Herzum and Sims (81), the term "component"is used in many different ways by practitioners in the industry. However, some rather broad and general yet useful definitions are as follows:
"*An independently deliverable piece of functionality providing access to its services through interfaces.*"–Brown (82)

"*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*"– Clemens Szyperski (83).

"*A component denotes a self contained entity (a.k.a. black box) that exports functionality to its environment and may also import functionality from its environment using well defined and open interfaces. In this context, an interface defines the syntax and semantics of the functionality it comprises (i.e., it defines a contract between the environment and the component). Components may support their integration into the*

surrounding environment by providing mechanisms such as introspection or configuration functionality."– Michael Stal (84).

In essence, more precisely a component may be defined as:
"An independently deployable and compositional software element conforming to software architecture"

4.4 Using Software Components: Component Based System Development

The usage of a component in a software system includes using it to replace an out of date component to upgrade the system or a failed component to repair the system, adding it to the system to extend the system services, or composing it into the system while the system itself is still being built. Some researchers insist on a component being reusable during dynamic reconfiguration (85). Component Based System Development (CBSD) advocates developing software systems by selecting reliable, reusable and robust software components and assembling (shown in Figure 4.2) them within appropriate software architectures.

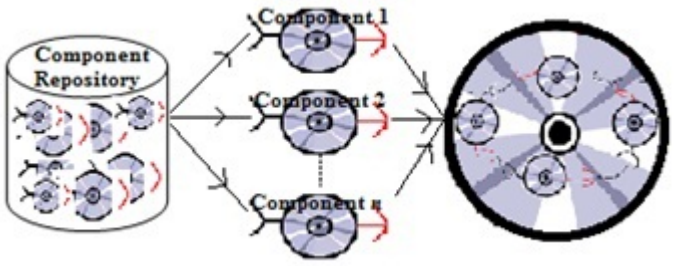


Figure 4.2: Component Based System Development

With the help of middleware technologies- a set of specifications or rules in the form of functions, which when incorporated into the code allows the software to be integrated with software developed using other platforms/languages (86). Example of such middleware technologies are COM, DCOM, CORBA, JavaBeans, EJB etc. Component Based Software Engineering (CBSE) is a process that aims to design and develop software systems using existing, reusable and adaptable software components as opposed to programming them. Hence, *CBSE shifts the emphasis from programming to composing software systems.*

4.5 Objectives of Component Based Software Development

The goal of component based development is to build and maintain software systems by using existing software components (77, 87, 88, 89, 90, 91). As said by Dr. Randall W. Jensen (92), "High customer demand, reduced software development budgets, and a competitive software market drive the need for reusable software". When correctly applied and implemented, developing software systems using Commercial Off The Shelf (COTS) software components promises benefits like increase productivity, shorten time-to-market, improve software quality, reduce maintenance cost, allow for inter-application interoperability, decreased level of risks, leverage technical skills and knowledge, and improve system functionality (93, 94, 95). As, software crisis may be loosely defined as the set problems associated with the software development process (62) i.e. quality, development cost and time-to-market of software, we may conclude that the indirect objective of CBSD are to alleviate software crisis. In addition, the particular objectives of software components are to:

1. **Develop Replaceable Components:** A component must have useful *replaceable property* i.e. easy to assemble and easy to disassemble.

2. **Increase Reusability:** Develop once and reuse several instances of the same over the period.

3. **Facilitating System Change Management and System Maintenance:** The plug and play feature of a component allows easy component composition and inclusion in the information systems.

4. **Enhancing Development Flexibility:** Components are an independent software element that can be designed and developed independently enhancing the development flexibility.

5. **Reduced System Development Time:** Reusing pre-developed existing components instead of new fresh development will reduce total development time.

6. **Reduced System Development Cost:** Reduced development time will result in significant reduction in total development cost.

7. **Improve Software Quality:** Ideally, a component is pre-tested for errors and quality parameters. Hence, using such pre-tested, high quality software components improves the quality of complete software systems.

8. **Reducing Project Risk:** From management perspective, if an asset's costs can be optimized through a large number of uses, it would then be possible for the management to expend more effort and allocate more budgets to improve the quality of software components. This in turn reduces the level of risk faced by the development effort and will undeniably improve the likelihood of success (96).

9. **Improve interoperability:** When systems are developed using reused components, they are expected to be more interoperable as they rely on common mechanisms to implement most of their functions (97).

10. **Increase System Learning for User:** Dialogs and interfaces used by these systems would be similar and would improve the learning curve of users who utilize several different systems built using the same components (96, 97).

11. **Ease and Efficient System Debugging:** As the components are previously tested, if any error occurs, must be during the integration. Hence, the domain and range for debugging is minimized and localized. This facilitates the debugging process quite easy and efficient.

4.6 Impact of Component Based Software Development

The CBSD approach includes improvements in: quality, throughput, performance, reliability and interoperability; it also reduces development, documentation, maintenance and staff training time and cost (1, 94). In this approach, due to inherent functional independence software is assembled from components can be autonomously deployed and, the productivity and performance of the development team can be improved (81). The impact of software reuse in system development is highlighted below:

1. **Impact on Productivity:** Although percentage productivity improvement reports are notoriously difficult to interpret, it appears that 30–50% reuse can result in productivity improvements in the 25–40% range. According to Lim (93), Hewlett-Packard software projects reported productivity increases from 6% to 40%

with the incorporation of CBSD. Further, Pitney Bowes in the USA which has been reusing components since 1996 documented tremendous savings in labour as the company is now able to achieve 500 human-weeks of development progress in only 200 human-weeks by using existing components and by purchasing others from component markets (98).

2. **Impact on Quality:** In a study conducted at Hewlett Packard, Lim (93) reports that the defect rate for reused code is 0.9 defects per KLOC, while the rate for newly developed software is 4.1 defects per KLOC. Henry and Faller (97) reported that, for an application that was composed of 68% reused code, the defect rate was 2.0 defects per KLOC—a 51% improvement from the expected rate, had the application been developed without reuse. They further report a 35% improvement in quality by component reuse in system development. They again suggested that, the quality of information systems developed using this approach will also have fewer bugs and defects if compared with newly built-from-scratch systems.

3. **Impact on Time-to-Market:** The STG division reports that the same development effort using the reusable work product required only 21 calendar months compared to an estimated 36 calendar months had the reusable work product not been used, a reduction of 42 % (93).

4. **Impact on Cost:** Apart from productivity gains, component reuse allows organizations to reduce the critical path in the delivery of software systems, reducing the time-to-market and begin to accrue profits earlier. Study shows that there has been reduction in product cost up to 75–84% as a result of reuse (99).

4.7 Industrial Practices on Software Components

"The use of commercial off-the-shelf (COTS) products as elements of larger systems is becoming increasingly commonplace. Shrinking budgets, accelerating rates of COTS enhancement, and expanding system requirements are all driving this process. The shift from custom development to COTS based systems is occurring in both new development and maintenance activities. If done properly, this shift can help establish a sustainable modernization practice." – SEI COTS-Based Systems Initiative (100).

Because the potential impact of reuse and CBSE on the software industry is enormous, a number of major companies and industry consortia have proposed standards for component software. In recent years, component technologies have been well developed, such

as Enterprise Java Beans (EJB) of Sun (101), CORBA Component Model (CCM) of the OMG (102), and Component Object Model (COM) of Microsoft (103) and are discussed below :

- **SUN JavaBeans Components :** The JavaBean (101) component system is a portable, platform independent CBSE infrastructure developed using the Java programming language. The JavaBean system extends the Java applets to accommodate the more sophisticated software components required for component based development. The Bean Development Kit (BDK) encompasses a set of tools to facilitate the CBSD approach.

- **OMG/CORBA :** The Object Management Group has published common object request broker architecture (OMG/CORBA) (102). An object request broker (ORB) provides a variety of services that enable reusable components to communicate with other components, regardless of their location within a system. When components are built using the OMG/CORBA standard, integration of those components without modification in a system is assured if an Interface Definition Language (IDL) is created for every component.

- **Microsoft COM :** Microsoft has developed a Component Object Model (COM) (103) that provides a specification for using components produced by various vendors within a single application running under the Windows operating system. COM encompasses two elements: COM interfaces that are implemented as COM objects and a set of mechanisms for registering and passing messages between COM interfaces. From application point of view, "the focus is not on how implemented, only on the fact that the object has an interface that it registers with the system, and that it uses the component system to communicate with other COM objects (104)."

4.8 Conclusion

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. Developing software using Commercial Off-The-Shelf reusable component is known as Component Based Software Development (CBSD). Informally, application of Software Engineering principles and practices in CBSD is known as Component Based Software Engineering (CBSE). CBSD

qualifies, adapts, and integrates software components for reuse in a new system. In addition, often engineers need to develop additional components that are based on the custom requirements of a new system that are unavailable from component library. CBSD offers inherent benefits in software quality, developer productivity, and overall system cost, but yet many roadblocks remain to be overcome before the CBSD is widely used throughout the industry. We may conclude that, component based development is the future development process to cater the present software crisis. Industries must follow this development practices, built and enlarge their component library for future reuse. At the same time industries must train their developers to encourage and practice the component based software development process and need to set up appropriate facility centers for supporting CBSD process. Despite lots of potentialities, there are still lots of issues that are to be explored for being the CBSD successful. Hence, researchers have to take the responsibilities on their shoulder to resolve these issues and make CBSD a successful software development approach.

Chapter 5

Investigating and Analyzing the Desired Characteristics of Software Development Lifecycle (SDLC) Models

5.1 Introduction

Several Software Development Lifecycle Models (SDLCs) are in existence (18, 19, 20). Over the time different people have proposed different models to meet the industrial demands. Any SDLC process model should be a repeatable, clearly documented, highly effective and must be based on the best industrial practices. However, the traditional SDLC process models provide very insightful theory and helpful best practices, but do not provide the practical details for daily application. As a result, statistics (62) shows that SDLC process models are rarely used by organizations for the purpose they are designed and developed for. Another primary reason for not using these models is due to lack of their suitability for real life projects - which led to software crisis. While investigating the reasons for unsuitability of these models, it is identified that we lack well defined characteristic parameters for any SDLC model. Without applying the process model in real project, we do not have adequate metric to analyze the suitability and goodness of such models. Davis et al (105) has proposed a strategy long back in 1988 for comparing alternative SDLCs only based on the ability to satisfy user needs and reduced life cycle cost. In this work, we have investigated, identified and analyzed the features of any

Based on the publication in the "*International Journal of Advanced Research in Computer Science and Software Engineering(IJARCSSE)*", ISSN: 2277 128X, 2(3), 311–315, 2012.

SDLC process model in general which may further be used for characterizing any SDLC process model for its suitability.

5.2 Objectives and Goal

The objective of this work is to identify the different characteristics of a good software development lifecycle model. Given these characteristics, one can judge, evaluate, predict and select the best SDLC model suitable for real projects. The outcome of this work then can be used while designing and developing new process models too. Using such metric, one may evaluate a model for its suitability, applicability and predictability of success for any project. Moreover, these can be used to design the quality, suitability and predictability metric of any process model. The outcome of this research may further be used to develop new SDLC models according to the need of the industry and even may be used while designing any new process model. Hence, the goal of this work is to develop the foundations for SDLC metric.
We shall use the term software development lifecycle process model and process model over the paper interchangeably.

5.3 SDLC Process Models and Objectives

There are numerous examples of disasters that had been caused by software failures. As the computerization of the society continues, the public risks of poor quality software will become untenable unless orderly steps are taken to improve the software processes (106). Any SDLC process model has three primary business objectives (107):

- Ensure the delivery of high quality systems,
- Provide strong management controls over the projects, and
- Maximize the productivity of the systems development team.

Further, these objectives can be broadly categorized from the following two perspectives:

1. **The Technical Perspectives :** While building a system, there remain many technical activities and issues including system definition (analysis, design, coding), testing, system installation (e.g., training, data conversion), production support (e.g., problem management), defining releases, evaluating alternatives, reconciling information across phases and to a global view and defining the project's technical strategy etc. to be resolved.

2. **The Management Perspectives :** When we plan to develop, acquire or revise a system, we must be absolutely clear with the objectives of that system. The objectives must be stated in terms of the expected benefits that the business expects from investing in that system. The objectives should exhibit the expected return on investments. To achieve the project objectives and goal many management related issues have to addressed and resolved. The primary management activities includes setting priorities, defining objectives, project tracking and status reporting, change control, risk assessment, stepwise commitment, cost/benefit analysis, user interaction, managing vendors, post implementation reviews, and quality assurance reviews etc.

All the above objectives irrespective of technical or managerial, has to be achieved through some SDLC process model if possible. But, unfortunately not all the available process model does address these issues efficiently.

5.4 Desired Characteristics of DLC Models

In order to meet the project objectives and goal, SDLC have to satisfy many specific requirements i.e. being able to support different types of projects and systems of varying scopes, supporting both the technical and management activities, being highly usable, and providing guidance on how to execute and install it for solving real life problems and many more. In the following section we are going to identify, enlist and discuss briefly some primary characteristics that are expected from any SDLC process model. As the degree of importance of these characteristics does vary from project to project, here we just enlist these characteristics alphabetically.

- **Change Management:**
  Requirement changes are often necessary, frequent and inevitable. The drivers of requirement changes may be customer demand, technical demand, competitive demand or even governmental or business policy demand. While occasional changes are essential, historical evidence demonstrates that the vast bulk of changes can be deferred and phased in at a subsequent point. To develop quality software on a predictable schedule, the requirements must be established and maintained with reasonable stability throughout the development cycle. Changes will have to be made, but they must be managed and introduced in an orderly way. Hence, change management is a critical part of any SDLC model. As requirements are

changed frequently, there is a need of streamlined flexible approach to manage these requirement changes within the SDLC model. Although requirement change management and SLDC are not mutually exclusive but the change management activities occurs throughout the development process. Further, cost of adopting changes is higher after the completion of the development. Hence, the objective should be to limit the change management activities within the initial development period as much as possible. If change is not controlled, orderly testing is impossible and no quality plan can be effective.

- **Concurrent and Parallel Development:**
  If high cohesion and low coupling modular system design is possible, then concurrent, distributed and parallel system development activities can be employed. This can improve productivity, timely system delivery while reducing total development cost and optimal usage of available resources.

- **Coordination among project stake holders:**
  A software project is not an individuals' job, but a collective effort towards the common goal. The success of software development projects depends on carefully coordinating the effort of many individuals across the multiple stages of the development process. Coordination —- long recognized as one of the fundamental problems of software engineering has become ever more challenging. This has led to a growing body of work on coordination in software development (108). With the rapid advancement of Information and Communication Technology (ICT), the location or center specific project development barrier are being diminishing. For optimal resource utilization, often project development activities are distributed over different development centers. Further, more the people are involved in one job, more the chances of misunderstanding and communication gap. Hence, if large numbers of people are involved and scattered over different development centers, the process model must provide mechanism for better coordination among the project stakeholders.

- **Cost of Life Cycle Implementation:**
  Additional costs and overheads are the primary barrier in process model implementation. For this reasons, many organizations do not implement or follow any process model. An ideal process model implementation should be economic, easy and justifiable. It should not require additional, special application or software purchase to effectively perform the process implementation or ongoing process management. In

addition, it should be easily automated utilizing any internal process management software tools currently being used within an organization.

- **Customer Involvement and Interaction:**
  In most of the common process model, there is no direct communication among customer, development team or project management team throughout the development process. In traditional models, management plays the vital role of bipartite body who works as the communication channel and messenger in the communication between customer and development team. As a result always there remains some communication gap and some missing or hidden information yet to convey to the development team but with the management. As a result, often proper requirements remain unspoken or hidden to the development team. Even conveyance of information might cause a loss of knowledge, as great amount of data remains with its carrier and never get handed off to others (109). Some interviewees suggested that the lack of direct contact between the development team and the customers could encumber the process of specifying requirements for the future. In turn handoffs among functions can cause delays and increasing risks of information being misunderstood (110). According to interviewees, level of details is varying depending on representatives between customer and developer. As a result, the developed system is frequently not satisfactory or even lead to project failure. Allowing direct communication of development team with the customer during the entire development process could eliminate project completion time and recourses consuming non value-added-action in form of handoffs, therefore waste (111). Hence, user or customer involvement during all phases of the project development is very important for project success and must be supported by any process model.

- **Proper and Sufficient Documentation:**
  Documentation has two ways of influencing the development process. Firstly, it makes the development process easier to understand. Secondly, documentation enables easy system maintenance, which can be linked to one of the principles in lean philosophy – knowledge sharing. But, unnecessary documentation can be addressed as waste. Any process model must enforce to develop the necessary document concurrently with the development process, while must avoid producing unnecessary documents to prevent miss utilization or waste of resources as in the case of agile development philosophy.

- **Early Defect Removal:**
  In case of any error or defect, if possible, it is always better to remove or rectify them in the earliest phases of the SDLC process model. Hence, the process model must focus on identifying the errors in the same or closest phase of the SDLC process to avoid or reduce the redo-work and cost. The best way to identify errors is to perform a close and effective verification after each and every phase and to set specific predefined phase entry and phase exit criteria effectively.

- **Easy to Execute:**
  Not all process models are easy to execute. Some process execution may require additional focus than the other. But often degree of easiness in execution may affect the other evaluation criteria of the process model. Always neither all easy process models are bad nor are all complex process models good. Hence, the tradeoff must be resolved depending on the suitability project demand.

- **Effective Management and Control:**
  Most of the existing traditional SDLC process models don't involves management team directly with the development team. Hence, the project management team does not have direct communication with the development and associated members. The management just remains as a silent intermediate communication body. Thus, proper management observation and control is hidden in the development process. As a result, the development process lacks proper management supervision and controls. In addition, the project has to suffer from resource shortage, risk handling, coordination and many other conflicts and problems. To overcome these problems, a direct involvement of project management team with the development team is necessary and important. The software development process must be under statistical control of the project management team to produce consistent and better result through process improvement.

- **Focus Towards Goal:**
  The project objectives and goal must be well defined and specific. The process model should view software development within the context of the larger system level definition, design, and development. Further, it should recognize both the potential value of opportunity and the potential impact of adverse effects, such as cost overrun, time delay or project risks according to the system and project specifications. Hence, any process model must reflect the project scope, objectives and goal consistently during the development process.

- **Incremental:**
  Software development project may be divided into distinguishable cascading phases. Before starting a phase, it may require a defined set of inputs from its immediate previous phase. The incremental methodology maintains a series of such phases. However, in the design phase development is broken into a series of increments that can be implemented sequentially or in parallel. The subsequent phases do not change the requirements rather build upon them towards the project completion. The methodology continues focusing only on achieving the subset of requirements for that development increment and continues all the way through implementation. Increments can be discrete components, functionalities, or even integration activities. Hence, through the incremental methodology, quantifiable partial solutions may be given to the customer without waiting for the entire project to be completed. The subsequent partial system may be developed in parallel to the already developed and operational partial system that may be further integrated when the second incremental development is available. The incremental process increases the degree of customer satisfaction and product quality as the defect of the delivered partial system may be identified while at operation and necessary changes may be incorporated immediately and deliver with the next increment.

- **Iterative:**
  In iterative process, the development begins by specifying and implementing the partial software requirements available at the moment without waiting for the complete or full software requirements specification. Further, these partial requirements can be reviewed in order to identify additional requirements and necessary modifications are made. This process is then repeated to implement the newly identified and specified requirements producing a new version of the software at each such iteration of the process model. This allows the project team to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the software in use. Iteration enhances the ability of the project management team to efficiently address the requirements of stakeholders and to complete, review, and revises phase activities until they produce satisfactory results. The product is defined as completed when it satisfies all of its requirements.

- **Distributed or Multi-Site Software Development:**
  Recent advances in information technology have made Internet based collaboration much easier. It is now possible for a software team to draw on talented developers

from around the world without the need to gather them together physically. To solve the problems like team relocation and project delay, developing software at multiple sites has been considered these days. Besides the obvious advantage of being able to tap into a much larger pool of human resources, experts working together from two or more locations can actually yield better outcomes (112). In such cases, software managers have to be able to manage these distributed teams. They need to define sharper processes, tracked, overseen and ensure that they are followed.

- **Quick Implementation:**
  A predefined solution that does not require organizations to start from the very initial level would allow organizations to exponentially cut down the time required to fully complete a process implementation effort. A process blueprint solution would drastically reduce the time and cost associated with traditional process improvement by laying an effective foundation for SDLC organizations to build upon.

- **Phase Length and Cycle Duration:**
  Phase length and Cycle duration should be optimal. It is well known that long cycle and phase duration is one of the primary reasons for project failure. Further, long phase and cycle duration makes the performance measurement task more difficult. If the cycle and phase duration is long, there may be problem with resource scheduling and may promote unoptimized utilization of resources which may affect the project cost, quality and schedule. Another source of risk resides in the relatively long stages, which makes it difficult to estimate time, cost and other required resources for project completion. Further, if the cycle duration is more, the delivery of incremental and partial system delivery will get delayed that may decrease customer satisfaction. Hence, the process model must be designed in such a way that the duration of each cycle and length of each phase must be small.

- **Predictability:**
  Any process model should be predictable. That is, cost estimates, schedule and quality commitments would be met with reasonable consistency, and the quality of the resulting products would generally meet the users' needs (109). As money, time, people and many other resources are involved, and at the same time quality and customer satisfaction are prime concern, before starting the project we need to predict the future outcome from it. If the outcome is not favorable, carrying out the project is nothing but waste of resources and gaining loss! In addition, the

process model should ensure that we can produce desired functions with higher quality using optimal resources in lesser time in a predictable manner. Thus, the process should have a predefined level of precision to facilitate a complete, correct and predictable solution.

- **Process Tailoring:**
  There may be different kind of projects and situations when no standard process is applicable. In such cases, the process model should provide the flexibility to adopt itself with the project and situational demand maintaining the integrity and consistency of the process by permitting tailoring of the standard process. Hence, tailoring is the process of adjusting the standard process to obtain a process that is suitable for the project need and situation (113). Thus, process tailoring facility makes the process model more flexible and adaptable.

- **Progress Measurement:**
  To evaluate the project performance and management control progress measurement of the project work is very important. For this purpose, milestones need to be set at regular intervals. Otherwise the project may suffer from 99% complete syndrome (19, 20). Effective and proper project measurement is only possible when the development process is under statistical control (114).

- **Prototyping:**
  Prototyping is necessary when it is very difficult to obtain exact requirements from the customer at the beginning of the project. Given the prototype of the system, user keeps giving feedbacks from time-to-time and according to the feedback necessary modifications are incorporated in the proposed or to be developed system. By doing these, the hidden, unidentified user requirements may be discovered during the initial phases of the system development process. By doing so, project failure risks may br reduces while improving the degree of user satisfaction and system quality.

- **Quality Control:**
  Lack of quality assurance during the different phases of the development process is often a potential source of risk. Validating the product is restricted to a single testing phase lately in the development process. Hence, the testing phase is the highest risky phase, since it is the last stage wherein the system is put as a subject for testing. Thus, all problems, bugs, and risks are discovered too late when the

recovering from these problems requires large rework which consumes time, cost, and effort. Milestones and deliverables can be setup or specified for each step of the project. Each module of the project is thoroughly tested before the beginning of another module. Project requirements are measured against the actual results.

- **Reliability:**
  Any process model must ensure development of quality reliable systems. A potential source of risk resides in the relatively long stages of any process model, which makes it difficult to estimate, time, cost, and other resources required to complete each stage successfully. In general, if incremental model is followed, the partial working systems are delivered periodically. It increases the reliability of the process model as reliability of the system does increase gradually during each partial product delivery to the customer.

- **Repeatable:**
  A SDLC process model should be repeatable i.e. the process should be repeat in case of projects which are similar in type or belongs to similar domain. A repeatable process reduces the cost of process model implementation as it is well known, learned and experienced to the development team. Hence, repeat process will reduce the project risks and cost. The quality of the system will be better as the outcomes of the phases are predictable.

- **Reuse:**
  The advantages of reuse in developments are now well established. Studies show that reuse had great impact on productivity, cost, quality, time-to-market and customer satisfaction (115). But very few process models like BDIDGE (62) are designed keeping the view of reusability in mind. Hence, as reuse has potential advantages, support of reusability is an important desired characteristic for process model in recent days.

- **Risk Management:**
  Risk is commonly defined as a measure of the probability and severity of adverse effects (116). Risks are an inherent part of any project which must be managed in advanced (if possible) or during the development process. As all projects inherit some risks, it is desired that the process model should provide adequate scope for risk management. Project risk may be related to project staffing, resources, schedules/budgets, technical, requirements changes etc. The SDLC model must

focus on the risk associated with the project continuously so that the management can take necessary control measure to prevent project failure. To manage risks, any process model must provide direct control over the project by project management. But, it has been seen that a very few model i.e. spiral and BRIDGE (62) provides such direct management support to the development process.

- **Scope:**
  Client demands never ends! The more you provide, the more they demand! Hence, if the scope of the project is undefined, satisfying customer is just a dream. The process should clearly mention what is desired to be produced and the developed product should be comparable to the defined requirements. Thus, the process model should have its specific scope as well as must limit the scope of the project. Scope of the process model bounds the suitability of the process model for different types of systems and projects.

- **Security Assurance:**
  Effective security is incorporated at the onset of a project. If it is included as a requirement early in the system development and/or acquisition process, it typically results in less expensive and more cost effective security. Waiting to integrate security until later in the process usually results in interoperability issues and increased cost. Integrating security into the SDLC begins with being able to articulate the security properties desired within the system. This process is typically cyclical in refinement beginning at the top level and drilling down into what will eventually be security specifications. There are many ways to express the high level security requirements i.e. ISO 15408 and others.

- **Separation of Concern in Different Phases:**
  As the development process is divided into different phases with distinct objectives, hence the concern of different phases should be clear cut and well separated. Otherwise, there may be chaos during progress measurement, quality and other management problems. Without separation of concerns in different phases, it will be tough to set milestones effectively, in turn that will make the progress measurement task difficult.

- **Simplicity and Flexibility:**
  Neither all simple things are better, nor are all complex things bad! The process model should be simple to understand, easy to follow and well manageable. More

the process model flexible, it is easier to manage and follow for execution. As change is inevitable due to specific nature of system projects, flexibility to accommodate changes is a basic need from a process model.

- **Software Process Improvement and Feedback:**
  Software Process Improvement (SPI) is an approach to designing and defining new and improved software process to achieve basic business goals and objectives i.e. increase revenue or profit, and to decrease operating costs by manipulating or changing the software process. The objectives of software process improvement (SPI) are to process produce products according to plan while simultaneously improving the organization's capability to produce better products (117). Perfection of any process is done via constant improvements. During the project execution or at the end, the team members give feedback in the form of reports suggesting different ways to improve the development process for the next iterations or future. One of the conditions required to improve the development process is to have the input from previous cycles so that the team's opinion and experience gathered during previous phase can be disseminated among other teams. This supports the process improvement throughout the whole organization, by adopting one of the core lean principles such as knowledge sharing, which in this case drives forward another lean principle – perfection of the process (118). The benefits of SPI include increased customer satisfaction, productivity, quality, cost saving and cycle time reductions.

- **Statistical control:**
  As Lord Kelvin said: "when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science" (109). Statistical control means that if the work is repeated in roughly the same way, it will produce approximately the same result. Measurement is the primary principle behind the statistical control. A SDLC model should be under statistical control of project management team. Such a process model will produce the desired results within the anticipated limits of cost, schedule and quality. If the process is not under statistical control, no progress is possible until it is (114).

- **Suitability to Projects:**
  Not all process models are suitable for every type of projects. Some process models are suitable for large projects, while some may be better for small projects. But, there are some flexible process models which are suitable for different types of projects. Any process model should not be suitable to only a specific type of project or project scope. The model should be designed in such a way that it should support different types of project with their varying scope through process tailoring. Hence, suitability to projects may be considered as an evaluation criteria for a process model.

- **Support to the Modern Tools and Technologies:**
  Over the time, significant developments are made in the field of new techniques and methodologies. Those are to be incorporated, accommodated and supported in the process models to make it a sustainable for modern software development. If a process model fails to accommodate these new technologies, they gradually become obsolete and useless. For example, during last few years there has been significant development in the field of CASE tools for project management, configuration management, software design, modeling and many more. It is a proven fact that usage of CASE tools(119) increases the product quality and reduces the total project development cost and time to market. Hence, the process model should be designed to support and utilize the CASE tools.

- **Usability:**
  The usability requirement addresses the various ways in which the SDLC will be used by the team members easily, efficiently while at use. Usability is a composite property of a process model. It is a composition of five attributes i.e. i) Learning ability, ii) Efficiency, iii) User retention over time, iv) Error rate, and v) Satisfaction (120). Any process model should incorporate these usability attributes. Many usability process has been proposed by several people i.e. ACUDUC (Approach Centered on Usability and Driven by use cases) by Seffah et al (121), Usability Engineering Process Model (UEPM) by Granollers (122) and Xavier Ferre (120) has proposed an integrated model of usability and different SDLC activities to integrate usability especially in different SDLC models. Adoption Centric Usability Engineering (ACUE) facilitates the adoption of usability engineering methods for software engineering practitioners and thereby improves their integration into existing software development methodologies and practices (123).

5.5 Conclusion

An important initial step in addressing software problems is to treat the entire development process as a performable, controllable, measurable and improved process as a sequence of tasks that will produce the desired result. Any fully effective software process must consider the interrelationships of all the required tasks, tools and methods, skills, training and motivation of the people involved. In general, any process model must bear the properties that are investigated, identified and specified in this chapter. At the same time, an ideal SDLC process implementation should be quicker, cost effective and easy to implement and follow. It should also be stakeholder and project team friendly. Finally, the ideal process model should address the top challenges experienced by project managers. Although the degree of importance of these individual characteristics may vary from project to project and depends on situational demand, but we recommend that all the above discussed characteristics should be beard by any process model to suit for the modern real projects.

Chapter 6

BRIDGE: A Model for Modern Software Development Process to Cater the Present Software Crisis

6.1 Introduction

Now a day, computers running with special purpose application software are being used as an extensive aid to solve complex problems almost each and every place starting from gaming to engineering, industries applications, scientific research and different allied fields. These special purpose software are sometimes unique and distributed in nature with higher degree of complexity. Developing such complex software is not so easy because of the different constraints. Our existing software models do not provide adequate flexibility to be applied for such large and complex projects. So we must have a better software development process model that will help to overcome these challenges.

6.2 Usage of Different Process Models: A Survey Report

The result of the survey carried out by Dr. Jon Holt (124), related to current practice in software engineering reveals the percentage of usage of different types of software development lifecycle models (SDLC) in practice. The result is shown below in Figure 6.1. Although different organizations do use different lifecycle models, but from the above data it is clear that a large part of industries (22%) do not use any lifecycle model

Based on the publication in the "*Proc. of the IEEE International Advance Computing Conference(IACC 2009)*", 494–500, 2009. (Also available at IEEE XPLORE) [DOI: 10.1109/IADCC.2009.4809259]

at all! The BIG question here is why these organizations do not follow any life cycle model?

The probable answer is, either no lifecycle model is suitable for their projects or they

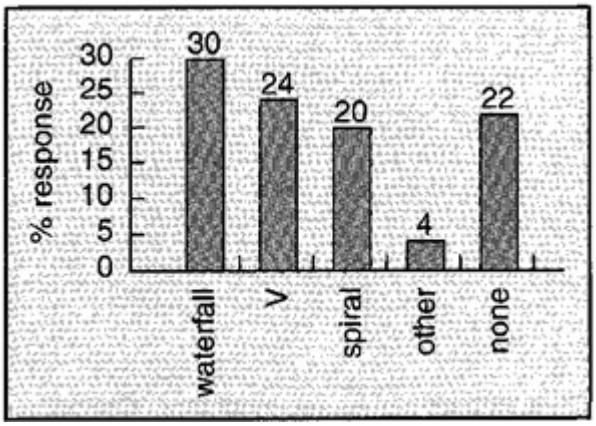


Figure 6.1: Use of Different Process Models (in %)

don't find it useful. In either case, it means the existing models lacking suitability. Hence, we need to improve the suitability of these models so that it can be used in practice.

6.3 Characteristics of good Software Development Process Model

Any software development process model should have the following characteristics (125) for quality software development:

1. The project goal reflection i.e. the process model must reflect the project development goals.

2. Predictability i.e. it must be able to forecast the output of the project following the model prior to project completion.

3. Support testability and maintainability i.e. the process model must focus on reducing the cost, effort of testing and maintenance.

4. Support change i.e. the process model must handle the necessary changes.

5. Early Defect Removal, because the delay in error detection increases the costs to correct them.

6. Process improvement and feedback i.e. each project done using the existing process model must feed information back to facilitate further process improvement.

7. Quantitative progress measurements i.e. the process model at any point must give a quantitative measurement of the progress attained.

8. Support of process tailoring in special situations at necessity.

6.4 Nature of Modern Software Projects

The earlier software projects were of limited scope with relatively less complexity and smaller size. In contrast, the modern software has wider scope, higher degree of complexity and larger size with better quality, portability and scalability requirements. Sometimes, the modern software has to work with some existing legacy system. Developing such system are more challenging because of the interoperability and dependency factors. The modern real-time systems have lots of critical issues such as time and space complexity requires to be addressed. Tremendous hardware development rate has brought us towards the System-on-Chip (SOC) era. In such systems, the software has to work in coordination with the particular hardware. Developing such systems are more critical because of the hardware constraints. As result of advancement in network technology, more often systems are becoming web based and distributed in nature. In conclusion, the modern software are different in various respects from the earlier software.

6.5 Modern Software Crisis

Software Crisis may be loosely defined as the problems associated with the software development process. Among a lot, a few critical software crisis with modern software development are listed below (18, 126):

1. Larger size

2. Increasing complexity

3. Higher development cost

4. The delivery challenges i.e. late system delivery

5. The trust challenge i.e. how much can we trust on system operations?

6. Incorrectness i.e. not satisfying the client needs exactly
7. Poor quality
8. Poor productivity
9. The heterogeneity Challenges i.e. inter-system coordination problem
10. Demand of reusability
11. Modularity
12. Maintainability
13. Integration problem
14. Scalability
15. Portability
16. Change Management
17. Risks associated with software development

6.6 Trends in Modern Software Development

Recently, lots of new approaches are being used at practice to overcome the modern software crisis. Some recent trends in modern software developments are listed below:

- Component based software development
- Software reuse
- Aspect oriented software development
- Service oriented software development
- Multi-Tired Software Design
- Object Oriented Software Development
- Standards practices
- Use of CASE tools

6.7 Reasons for failure of Traditional SDLC Models: The Shortcomings

After analyzing the existing SDLC models, the shortcoming of these models may be broadly summarized as follows:

1. Non-involvement of the client over the entire project development
2. Lack of better understanding of the system requirements
3. Lack of communications among the team members
4. Lack of project management controls over the entire development period
5. Overlooking verification activity
6. Insufficient documentations
7. Lack of configuration management
8. Non importance to component based software development and
9. Poor support of component reusability

Directly or indirectly, the above reasons are the real causes of the various software crises. I have tried to address these causes of software crisis in my proposed model discussed shortly.

6.8 Need of Modified Process Model

Although, tailored traditional software development process models are being used since a long time, but these are not good enough at practice. Hence, we are in search of a new software development process model that will adopt and encourage these modern practices. In the forth coming section a rather novel software development process model-BRIDGE, is proposed and discussed that attempts to encourage the modern software development trends. As well said by David Norton, research director at Gartner “I do not feel waterfall development was bad. It's given us a lot of software over the last 30 years, but I think its time is up (127)”.

6.9 BRIDGE: The Model for Modern Software Development Process

After analysing the importance of all the recent software development trends, an attempt is taken to develop a rather new and novel software development process model that adopts the modern software development trends and practices. The so named BRIDGE model is the result of such an attempt, which is elaborated over the following sections. The schematic diagram of the BRIDGE model is given in Figure 6.2.

6.9.1 BRIDGE Process Model Description

Unlike the other process models, the BRIDGE model consists of several phases with distinguished objectives that are discussed in the following section briefly:

Phase 1: Requirement Analysis, Verification and Specification
The objective of this phase is to identify the exact requirements from the client using different techniques and to specify them in a document for future use after verification. During requirement gathering, the analyst extracts the system requirements from the client. In practice, it is really a tough job for the analyst to extract the requirements from the client, as the clients are unable to identify and express the exact requirements prior experiencing the system practically. The gathered requirements required to be analyzed for removing the redundancy, incompleteness, inconsistencies, anomalies etc. This phase is often called the requirement analysis phase. Finally, the verified requirements are to be specified in a document called Software Requirements Specification (SRS) and stored for future use. This phase is often called requirement specification phase. This SRS document may serve as the agreement document between the client and the company and becomes the baseline for proceeding to the next phase.

Phase 2: Feasibility Analysis, Risk Analysis, Verification and Specification
The objective of this phase is to analyze the suitability of the project in respect to different project attributes to check the different suitability aspects among the alternatives. After carrying out the analysis, the optimal solution is selected. At this stage the project cost estimation has to carry out. The different feasibility i.e. economic feasibility, technical feasibility, operational feasibility has to carry out to manage the different system constraints. Some times, the result of the different feasibility analysis may contradict. In such cases, necessary changes, modification and/or negotiation may have to do in the

project upon consulting the client if the project is not canceled. Finally, after verification the result of the feasibility analysis has to be specified in a document called feasibility report and to be kept for future reference. Beside feasibility analysis, at this phase the different project risks have to be identified, analyzed and specified in the risk specification document.

Phase 3: Software Architecture Design, Verification and Specification
Once the project is confirmed, we must design the software architecture. Software architecture design is a high level design activity and relatively a recent trend in industries after understanding its importance. We may consider software architecture as abstract design of the complete system. The objective of software architecture design is to identify the subsystems, building blocks or the components of the system along with their communication interfaces expressing their external behavior to improve the project understandability and to communicate with the different stakeholders. The architecture design should reflect the functional requirements specified in the SRS document. Once the software architecture is designed, the architecture design must be verified to check whether it conforms the system requirements correctly or not. The verified software architecture design is specified in a software architecture design document (SADD). It must be clear that implementational issues are not considered while designing the software architecture.

Phase 4: Detailed Software Design, Verification and Specification
In this phase, the detailed design of the system has to be prepared conforming the software architecture designed during the last phase. Software design is basically a low level design activity keeping the implementational issues in mind. The objective of this phase is to prepare the modular design of the system that can be directly implemented using some programming language. The data structure and algorithms are also to be developed in this phase. The verified software design specified in a document named as software design document (SDD) that will be used in the other development activities later.

Phase 5: Patterns Identification, Component Search, Verification and Specification
In general, a system consists of a set of subsystems, so called components. If we analyze any problem, we may find some components common in different projects representing some general structures of a system. These common components are sometimes called

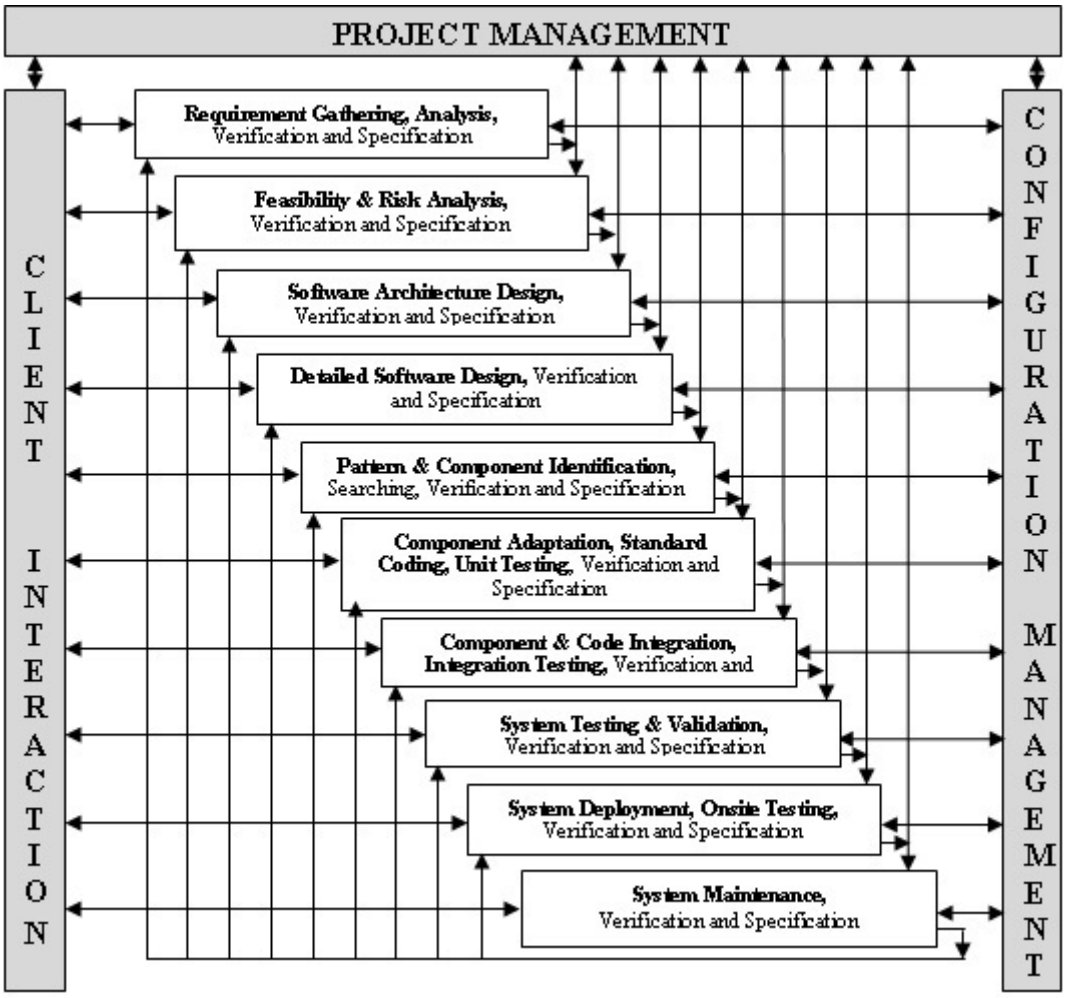


Figure 6.2: BRIDGE Process Model

patterns. The objective of this phase is to identify these patterns. But, to use these pre-developed components efficiently in our system, the system must be designed keeping this objective in mind and the designer should be well aware of the available components in the component library. From the architecture design, we must be able to identify the components and then it must be searched in the component library to find a suitable component match. Before moving to the next phase, we must verify the current phase properly and specifying in a document called component specification document (CSD) for future use.

Phase 6: Standard Coding, Unit Testing, Verification and Specification
All the components identified during the last phase may not be available in the component library. The objective of this phase is to write program code for the unmatched

components. Often, a few unmatched components may work as desired just with a suitable added interface. In those cases, the benefit analysis must be done to take the decision whether to develop the interface only or the unmatched components from the scratch. The unmatched modules must be coded properly following the standard coding guidelines and practices laid down by the organization itself or the available standard conventions as per the organization interest. These newly developed components must be tested thoroughly since these components are going to be used in several systems at different times. Such testing is called unit or component testing. The components taken from the component library together with the newly developed components should be sufficient enough to build the whole system. The newly developed components may be added in the component library for future use if it looks justifiable. After verifying and specifying the phase properly, next phase can be started.

Phase 7: System Building: Component Integration, System Testing, Verification and Specification

Once all the individual components are gathered, it's the time to integrate these to build the whole system preferably following the bottom up approach. Hence, the objective of this phase is to build the whole system by integrating all the components. However, it is not necessary that, after integrating the pre-tested components successfully, the integrated system will work correctly. Various types of problems such as type mismatch, number of parameter mismatch, return type mismatch etc. may arise. Hence, there is a need to test the integrated system at different level of integration. This is called integration testing. Now, the complete buildup system has to be tested thoroughly using the different testing techniques to check the correctness of system functionality. The testing at this topmost level is termed as system testing. After performing the different testing, the corresponding test report has to be prepared for use during system validation and maintenance activities. Finally, the phase verification is to be carried out prior moving to the next phase.

Phase 8: System Validation, Verification and Specification

Merry successful verification of the system doesn't ensure the fulfilling of all client requirements! By successful verification of the system, we can only ensure that whatever the functions are implemented in the developed system do work correctly, but does it mean that, all the function required by client are implemented in the system? No. The objective of this phase is to check whether all the functional requirements as specified in

the SRS document specified by the client are exactly included the system or not. There must be one to one correspondence between the functions in the SRS document and function supported by the system. Performing this activity is called system validation. Not only the system functionality but also the quality of the system has to be validated. Unlike the other phases, at the end, this phase to be verified and the outcome of the system validation activity are to be specified in a document called validation report and stored for further use.

Phase 9: System Deployment, Implementation and Specification

Once the system is validated, now it's the time to deliver the system to the client and implement the system at client site. Again, some more changes may be required to accommodate and adjust for proper functioning of the system. Delivering the system to client should not be taken as a formality! Ultimately you- the developers are not going to use the system, but the users definitely. Until the users are not able to use the system effectively and efficiently, developing the system remains purposeless. We must facilitate the user to understand and feel comfortable with the system at use. There are basically two tools for this purpose. First, the documentations and second, training. Necessary training has to be provided to all different categories of users within their operational scope. The user refers to the documents to solve problems at any point of time during the system use. The objective of this phase is to deliver, implement the system at client's work site and train the users, if necessary. After verification of the phase necessary documents are to be prepared and retain.

Phase 10: On Site System Testing Verification, and Specification

Although, system testing is completed prior to system implementation, but due to different environmental changes and other reasons, the system may not function correctly at the work site. Hence, after implementation, the system needs to be tested at work site too. This testing is called on-site system testing. The objective of this phase is to check the system performance at work-site. Finally, the on-site system testing report has to be prepared and to be retain after the phase verification. At this point, the current system is at work.

Phase 11: System Maintenance, Verification and Specification

Merry successful system implementation and functioning is not the end job. There is a well saying that no software is correct at all. Moreover, Lehman's first law related to

software says, "Software product must change continually or become progressively less useful" (126). Software Maintenance denotes any changes made to a software product after it has been delivered to the client. Maintenance is a continuous process over the software life cycle. The objective of this phase is to provide the post delivery services to the system for its desirable functioning. Maintenance support is to be provided to retain and improve the system quality over its lifetime. The maintenance may be of different types i.e. corrective maintenance, adaptive maintenance, perfective maintenance and preventive maintenance (19, 126). Finally the maintenance report is to be made periodically and kept for future reference. It should be clear that deliverables from any phase might be given as input to the other phases if needed.

Phase 12: Configuration Management

As we have seen, system requirements always change during system development and use. Accordingly, these changes have to be made in associated documents and dependable. Finally, these changes are to be incorporated into new version of the system. Hence, it is clear that, the deliverables from different phases are to be maintained for future use. As we have seen over the past discussion that, from each phase different documents are produced and need to be kept properly for future use. The patterns are even to be kept in such a way that, on demand we must be able to identify, search, and locate all these components for adaptation. Hence, we need to have an efficient document and component keeping system. If there is no proper management and control over these changeable particulars, then it is very tough to incorporate these necessary changes in the following version of the system. The means by which the process of software development and maintenance is controlled is called configuration management. The objective of the configuration management is the development of procedures and standards for cost effective managing and controlling the changes in the evolving software system aiming to keep track of all the important deliverables obtained from different phases.

Phase 13: Project Management

Unlike the configuration management activity, the project management activity has to be carried out in parallel with all the other software development phases. While developing software, we need to carryout some management activities that are part of project management. The objective of this phase is to perform the project management activities including project planning, monitoring, controlling, directing, motivating and coordinating.

6.10 Analysis of the BRIDGE Process Model

The in-depth study of the BRIDGE model discloses a lot of information that may be used to analyze the model. These are briefly discussed below:

6.10.1 Findings from the Study of BRIDGE Model

The findings from the BRIDGE model are listed below:
i) It involves the client over the entire development life cycle activities.
ii) It keeps continuous communication with the project management team.
iii) It explicit verification of individual phases.
iv) Separate software architecture design phase.
v) Separate system deployment phase.
vi) Separate on-site system testing phase.
vii) Supports components based software development.
viii) It emphasizes on standard coding.
ix) It considers configuration management as a separate activity.
x) It forces to specify all the phase deliverables.
xi) It explicitly instructs to validate the system.

6.10.2 Impact analysis of the findings from BRIDGE Model Study

In this section, impacts of the findings from BRIDGE model studies on the project goal are analyzed distinctly.

1. **Impact of continuous client involvement:** It is experienced that, as the system is more studied and analyzed over the time, the client specifies more new requirements. Satisfying these requirements, client satisfaction and software quality are improved with great impact on both project and organizational goal. Moreover, involving the client over the entire SDLC project risks can be alleviate up to a significant extent. By means of continuous client involvement, this model can embed the prototyping paradigm of software development.

2. **Impact of continuous project management team involvement:** The impact of involving the project management team over the SDLC model may facilitate effective project management activities such as project planning, progress monitor-

ing, project controlling, risk management, Motivation and individual performance analysis used for organizational and personal appraisal.

3. **Impact of explicit verification activity:** By verifying the individual phases indirectly the phase entry and exit criterion may be satisfied which reduces the error occurrence rate in the later phases. This may even overcome the well known "99% complete syndrome problem". Verification helps in early error detection and correction reducing total development cost having direct impact on software testing, quality control and timely product delivery.

4. **Impact of software architecture design:** Software architecture is the key framework better project understanding and communication with the various stakeholders. Software architecture has a profound influence on organization functioning and structure (128). Designing the software architecture has the direct impact on the software quality attributes such as performance, security, safety, availability, maintainability, scalability, productivity, cost, effort and timely product delivery.

5. **Impact of separate system deployment phase:** It directly maps the environmental view supported in UML. There is a very poor practice of considering system delivery as just a formality. Proper training must be given to the users for efficient and effective system use. More over it helps to handle all software crisis related to product deployment improving the software quality.

6. **Impact of separate on-site system testing phase:** The on-site testing helps to improve system quality and client satisfaction reflecting the long-term goal of the organization.

7. **Impact of component based software design:** The component based software design helps in achieving better software maintainability, reusability, productivity and quality reducing total development cost and effort.

8. **Impact of following standard coding:** Following standard coding practices and conventions have remarkable impact on better understanding of the code written by others reducing efforts in error isolation and system testing improving the maintainability, quality of the software. It does encourage good programming practices.

9. **Impact of configuration management activity:** Configuration management activities improve different documents and components management. It does facilitate component repository and reusability reducing total development cost and

efforts improving the software quality and increasing organizational assets simultaneously.

10. **Impact of document specification:** The different specified documents facilities better system understanding leading to ease error handling. These are the means of communication among teammates and stakeholders. It helps in reduction of testing and maintenance efforts.

11. **Impact of system validation:** System validation ensures correct system functionality by error detection achieving the goal of better quality software development. Finally, it increases degree of client satisfaction attaining long-term project and organizational goal.

6.11 Validating the BRIDGE Model in Support of Goodness Criterion

The proposed BRIDGE model does satisfy almost all the goodness criterion (125) of a good software development process. In this section, I discuss the supporting issues for validating this model against the individual goodness criteria.

1. **Support towards project goal reflection:** As per the definition of software engineering given by Stephen Schach (129), the goals of software project are:
   a. Developing quality software.
   b. Developing the software within budget.
   c. Delivery of the software within time.
   d. Satisfying customer requirements.
   By focusing on the phase verification and validation activities, and recommending software testing at different levels, this model reflects the goal of developing quality software. Again, specially performing economic feasibility analysis and involving the management over the process, the model reflects the goal of developing the software within budget. On stream, by involving the project management team over the entire process development model, this model puts focus on proper management control to follow the time constraints on the project development. Finally, by means of client involvement over the complete software development process, the BRIDGE model achieves the goal of customer satisfaction.

2. **Support of Predictability:** The software architecture is the best document to predict the different project parameters. Having a separate software architecture phase and risk analysis, this model achieves the predictability criteria.

3. **Support of testability and maintainability:** Emphasizing on component based software development and component reusability concept, this model highlights the testability criteria. In addition, designing the software architecture gives the foundation for meeting maintainability criteria with a separate phase related to software maintenance.

4. **Support towards change:** Designing software architecture and by supporting maintainability, this model achieves the change management criteria directly with consistent support from configuration management.

5. **Support of early defect removal:** By involving the customer over the entire development process, it is possible to detect errors at earliest and performing verification activity following each phase ensures early defect detection and removal.

6. **Support of process improvement and feedback:** During the configuration management activities, all the prepared documents and reports are stored. Project completion analysis report with the available documents and the reports from configuration database, can be used to judge and identify the activities needing process improvement and applying the same in the next project. Customer comments and recommendations can be used as the feedback for further process improvements.

7. **Support of Quantitative Progress Measurement:** Directly, each phase indicates a milestone towards the project completion. All the deliverables from various phases of this process model can be used to measure the progress of the work completed.

8. **Support of Process Tailoring:** Since the process activities are decomposed in several phases, at necessity, more than one phase can be combined and any phase can be further decomposed into sub phases or even might be dropped depending on the project characteristics. Hence, it may be concluded that, the BRIDGE model satisfies all the desired characteristics of a good software process model.

6.12 Suitability of the BRIDGE Process Model

This model can be used to both simple systems as well as complex systems. It supports the object oriented, component based software development paradigm. By process tailoring, this model also can be applied to develop any software projects that are directly unfit to the actual model. Hence, the suitability of the BRIDGE model for any modern software development is justified and may be recommended for any kind of software project development.

6.13 Limitations of the BRIDGE Process Model

Along with the strong suitability, this model has some limitations as pointed down below:

1. Non-considering the implementational issues.
2. Abstracts the different techniques to be used in different phases.
3. Required to be validated by industrial practice.
4. It doesn't consider professionals skill level.
5. The BRIDGE model seems to be complex.

6.14 Naming Significance: The BRIDGE

The schematic diagram (Figure 6.2) of the proposed model looks like a bridge. In a bridge, the entire load is on the bridge floor, but this load has distributed over all the pillars for its survivals. Directly the project pressure is on the "Project Management" and this pressure has to be distributed over "Client Interaction", "Configuration management " and other the phases indirectly- the pillars of the model. Keeping this point of view the name, BRIDGE, is given and justified.

6.15 Conclusion

After the complete analysis, it can be conclude that if the BRIDGE model is followed to any software project development, most of the software crisis may be overcome up to great extent delivering the fully functional system with better quality within time and budget achieving the true goal of any software project development.

Chapter 7

Achieving Agility through BRIDGE Process Model: An Approach to Combine the Agile and the Disciplined Software Development

7.1 Introduction

Many Software Development Life Cycle (SDLC) process models and approaches have been introduced till date i.e. Waterfall model, Spiral model, Prototype Model, Evolutionary development etc (19). But, rarely any of these models exactly fits as-it-is for modern software development projects (130). Hence, often instead of a single process model, most industry follows a sandwich or hybrid model i.e. mix-up of different models on demand basis (131). Despite of this, very often the customers are unhappy with the end product and many the projects even had to fail! Agile software development philosophy came out to be a successful approach to increase the project success rate up to significant extent while reducing the development time and cost. Agile software development (132) philosophy has been proved to be quite successful to increase the project success rate up to a significant extent. Despite of its success, many authors have criticized the agile approach as it often violates the basic theories, principles and practices of traditional software engineering. But the ability to meet client needs and the delivery of quality software products within estimated time is the significant benefits of agile development and is the key to its survival. Some of the additional benefits of agile process are increased productivity, expanded test coverage, improved quality,

Based on the publication in the "*Innovations in Systems and Software Engineering (ISSE): A NASA Journal*", ISSN: 1614-5046, 11(1), 1–7, 2014. [DOI: 10.1007/s11334-014-0239-x]

fewer defects, reduced development time and costs, delivery of better understandable and maintainable code, improved morale, better collaboration, and higher customer satisfaction as pointed out by Vijayasarathy (133). But as said by Boehm et. al (134, 135), neither the agile nor the traditional disciplined approach alone provide the ultimate approach. Further Hashmi et. al (136) says, there are ongoing debates whether the quality of the products of the agile approaches are satisfactory. In addition, Fritzsche et.al (137) and Theunissen et.al (138) mentioned, some projects e.g. safety-critical projects require standards to be followed when developing software. On the other hand, the two seem contradicting (139, 140), but several researchers agree that a software project needs both agile and discipline (134, 141, 142). New SDLC models are introduced at regular basis as new technology and new research results are needed to be accommodated over the time for modern project needs and suitability. This was the primary philosophy behind the introduction of BRIDGE software development life cycle process model by Ardhendu (62).

7.2 Scope of This Study

In this chapter we have just considered the theoretical aspects for the purpose of analysis and as evident. We didn't consider any real development situations or data for this instance. Presently we are running three experimental projects in our department in parallel to discover the impact of the BRIDGE process model. But, as it shall take more time to complete the projects and to evaluate the successfulness, we simply skipped this for the time being. The real development situations will be disclosed in the future work when the projects will be completed and the experimental results shall be available to us.

7.3 From Disciplined SW Development Approach towards Agile

7.3.1 Limitations of The Traditional Development Models

We have many traditional SDLC models i.e. Classical Waterfall Model, Iterative Waterfall Model, Spiral Model, RAD model etc. in our hand by the time, but a very few are really used in practice exactly as it is. There exist several criticisms about these traditional models. According to Nandhakumar and Avison (143), traditional methods

are too mechanistic to be used in detail. Truex et al. (144) pointed out that traditional SDLC models are more dogmatic and claim that traditional methods are merely unattainable ideals and hypothetical "straw men" that provide normative guidance to utopian situations. Further, Wiegers (145) noticed that, industrial software developers have become skeptical about "new"solutions that are difficult to grasp and thus remain not used. Baskerville et al. (146) claim that "to compete in the digital economy, companies must be able to develop high quality software systems at "Internet speed"—- that is, deliver new systems to customers with more value and at a faster pace than ever before". The primarily perceived limitations of the traditional development models are:

- There are too much work associated with documentation
- They are too sequential
- They requires too much of planning activities
- It does not show results until the end
- It engages stakeholders too late
- Delay in project delivery
- Increased Project Cost

For these reasons, the traditional software development approaches are rarely used in industries these days as they lack suitability for the purpose.

7.3.2 Agile Development : The Origin and its Necessity

Agile principles evolved to address the above criticisms and primary limitations of the traditional software development. Agile software development is neither a set of tools nor a single methodology. Rather, agile is a philosophy appeared as recommendations in 2001 with an initial 17 signatories. While the publication of the "Manifesto for Agile Software Development"(132) didn't starts the move to agile methods, which had been going on for some time, it did signal industry acceptance of agile philosophy. Further, Kieran Conboy (147) in his paper has brilliantly derived the functional definition of agile as "the continual readiness of an information system development method to rapidly or inherently create change, proactively or reactively embrace change, and learn from change while contributing to perceived customer value (economy, quality, and simplicity), through its collective components and relationships with its environment".

Agile was a significant departure from the heavyweight document-driven traditional software development methodologies in general used at the time. Agile software development stresses rapid iterations, small and frequent releases, and evolving requirements facilitated by direct user involvement in the development process. In this way, development of agile methods could be seen as cumulative methods built on existing traditional methods where the *good* parts are kept and the *bad* parts are omitted or modified.
As per the recent survey report on state of agile by Versionone (148), shows that 88% of the respondent organizations are practicing agile development, 52% of the projects are being developed following agile.
From the Agile Manifesto (132, 149) and the definition of agility cultivated by Conboy (147) we may extract and combine the following primary feature of agile methods to be put on focus:

- Individuals and interactions
- Working software delivery
- Customer collaboration
- Rapid system change incorporation i.e. responding to change
- Economic development
- Quality product development
- Simplicity
- Enhance knowledge from change incorporation.

In the next section we give the list of principles of agile development philosophy with brief outline.

7.4 Principle of Agile Development

There were twelve basic principles of agile software development as highlighted in the Agile Manifesto (132). The detail discussions of these principles are beyond the scope of this work. But for the purpose of justification and comparison, we combine and highlight these principles from Agile Manifesto (132) and as extracted from Conboy's (147) definition here in brief:

1. **Customer Satisfaction :** The highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. **Incorporation of Rapid System Change :** Agile methodology welcomes changing requirements even late in development. Agile processes harness change for the customer's competitive advantage.
3. **Frequent Working SW Delivery :** Deliver working software frequently- from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. **Continuous Cooperation of Client and Developer:** Business people and developers must work together daily throughout the project.
5. **Motivated Trusted Individuals :** Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. **Continuous Improvement-Arrangement of Face-to-Face Conversation:** The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. **Progress Measurement :** Working software is the primary measure of progress.
8. **Sustainable Development :** Agile processes promote sustainable development. The stakeholders, developers, and users should be able to maintain a constant pace indefinitely.
9. **Attention to Technical Excellence :** Continuous attention to technical excellence and good design enhance agility.
10. **Simplicity :** If the design and implementation are simple, testing is easier and more effective.
11. **Self-organizing Teams :** The best architectures, requirements, and designs emerge from self-organizing teams.
12. **Internal Assessment for Knowledge Enhancement :** At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

13. **Quality Assurance :** The organization must ensure the quality the system developed following the standard quality improvement practices.

14. **Economic Development :** Economic development should be achieved through optimum utilization of the resources through lean system development (147).

Some authors (149, 150) have done studies about scaling agile methods in regulated environments. But, practicing agile in regulated environments will imply many constraints to the agile process that may be either against the philosophy of agile or it could be equivalent to a traditional development approach. As pointed out by Fitzgerald et al (149), "Some of the essential characteristics of agile approaches appear to be incompatible with the constraints imposed by regulated environments". They further mentioned in the same paper that agile software development methods are faced with some fundamental challenges in regulated environments as a core characteristic of regulated environments is the necessity to comply with formal standards, regulations, directives and guidance. Further, as mentioned by Turk et al (151) that agile methods and regulated environments are often seen as fundamentally incompatible. Thus it may lead the agile process to be unlike a traditional process that is not the objective of agile always. The objective should not be to go away from the agile philosophy rather to be adhered to the Agile while achieving the advantages of disciplined approach. It is better not to tailor the agile methods to achieve the discipline necessary in regulated environments, but to optimally achieve the objectives of agile through some disciplined approach in regulated environment.
In the following section we review the BRIDGE model as for reference and then explain how to achieve agility through this model.

7.5 Introduction to BRIDGE Process Model

The BRIDGE process model was introduced and presented by Ardhendu (62) in IEEE IACC, 2009. Although, detailed discussion of the BRIDGE model is beyond the scope of this chapter, but we give the schematic diagram in Figure 6.2 of the model for the sophisticated readers just for reference with its primary properties.

The Primary Properties of BRIDGE Model
The primary properties of BRIDGE model are enlisted below (62):

1. It involves the client over the entire development life cycle activities.

2. It keeps continuous communication among the development team, project management team and client.
3. It enforces explicit verification of individual phases.
4. Supports Components Based Software Development (CBSD).
5. It enforces on standard coding practices.
6. It considers configuration management as a separate activity.
7. It forces to specify all the phase deliverables.
8. Separate software architecture design phase.
9. Separate system deployment phase.
10. Separate on-site system testing phase.
11. It explicitly instructs to validate the system.

7.6 Agile Development with BRIDGE Process Model

For detailed discussion about the BRIDGE process model and its primary properties introduced by Ardhendu (62), one may refer to Chapter 6 of this thesis. One may refer to Figure 6.2 for the schematic diagram of the BRIDGE process model.
In this section we are going to explore how the principles of agile can be achieved through BRIDGE process model. We consider all the principles of agile and discuss in brief how these are supported and can be achieved through BRIDGE model.

- **Achieving Customer Satisfaction :** The agile principles proposed customer collaboration for increasing customer satisfaction. In BRIDGE model this is achieved through continuous client interaction i.e. the left base pillar of the model.
- **Accommodation of Requirement Change :** The initial system requirements may not be mature enough at the very inception of the project. As the developers and client understand the system more and more over the development process, the requirements get refined, even may get modified over the time. As the client is engaged over the entire process, as soon as new requirement is discovered, it is accommodated within the same design or necessary modification is made in the

design and its subsequent phases by means of phase iteration. The iterative nature of the BRIDGE process model with the focus on component based software development facilitates accommodation of requirement changes easily in the system. Promoting system architecture design and component assembly based system development methodology (115), it is comparatively easy following BRIDGE process model to discard, add or modify system requirements.

- **Frequent Working SW Delivery :** As BRIDGE is an iterative process, hence we may start with the initial system requirements and design the initial working software with limited capabilities and deliver to the client at earliest. As the time moves, more and more requirements can be accommodated and delivered to the client in the subsequent software versions and variants.

- **Continuous Cooperation of Client and Developer :** This is one of the best features of the BRIDGE model. The client remains as an active member of the development process from the very inception till the end of the entire process. Hence, unlike agile in this model also there exist a close continuous cooperation between the client and developers.

- **Motivated Trusted Individuals :** Alike the client, the management unit also remain as a continuous part over the BRIDGE process model that enables the management to monitor the individual developers. Often, if needed the management may keep on motivating the developers to give their best and may take necessary actions to make them enough trust worthy for the organization. As a result, over the time the developers become more motivated trusted individuals for the organization. The management may investigate regularly about the need and working environment that can be allocated optimally so that the job can be done within time and budget while maintaining the quality.

- **Continuous Improvement-Arrangement of face-to-face conversation :** We know face-to-face conversation is the best way to convey and share information. Being an integral part of the process the management may arrange face-to-face conversation among the developers and even with the client for continuous improvement. As in BRIDGE process model management team is associated with the development over the development period, the management team may organize such events to promote continuous improvement easily.

- **Progress Measurement :** In general, before starting any phase it must satisfy some phase entry criterion. As each phase of the BRIDGE model has to go through a strict verification activity before starting the immediate next phase, hence all phase has to satisfy the phase exit criteria too. The phase entry and phase exit criterion are nothing but some intermediate milestones in addition to some other desirable constraints. These milestones may be even partial working software. Through these intermediate milestones, both the development and management team may assess and measure the progress periodically. Following the way, in BRIDGE model we may measure the progress of the project at regular interval.

- **Sustainable Development :** Continually evolving, growing, and changing is a natural phenomena of any organism- none of the other organisms can survive without it. The same thing can be applied to software also, just for analogy. Very few software is written once, installed, and then never changed over the course of its lifetime. As per Lehman's first law (126) regarding software, a software product must change continually or become progressively less useful. New requirement will get discovered over time, some old requirements may need to be modified or discarded for the products survivals and its enhancements. Hence, the system has to be designed and developed keeping the view of anticipating the future changes in mind. Following such type of development is what called sustainable development. But, unfortunately it is a rare practice as it may increase the product cost and other development burdens. Maintaining and enhancing software to cope with newly discovered problems or new requirements can take more time than the initial development of the software. Sustainable development requires a singular focus on a form of technical excellence that regularly provides useful software to customers while keeping the cost of change minimal. Under the umbrella of effective management, the project stakeholders can maintain consistent work pace and speed promoting sustainable development following the BRIDGE model as all of the stake holders works together in this model.

- **Attention to Technical Excellence :** In addition to design phase, the BRIDGE model has a specific software architectural design phase. Further, being the customer an integral part of the development process, technical excellence can be monitored by both the development and project management team. Continuous attention to architectural design, low level design and technical excellence of BRIDGE model enhances the agility.

- **Simplicity :** The notion of simple is very subjective and giving practical guidelines what is simple and how to accomplish that is impossible. At the beginning, requirements seem to be quite complex, but after the analysis and as the project development progresses they becomes clear. The developers should only implement features that have been agreed upon with the customers, nothing more. The art of maximizing the amount of work not done–is essential. Cockburn and Glass (152) warn that simplicity should not mean neglecting design by starting programming as soon as possible. This principle most strongly supports the design of high quality architecture and implementation. This principle further acknowledges that in programming it is more difficult to make simple design than cumbersome solutions. Following a disciplined and systematic approach makes any process simple. The system requirements should be simple and must specify the scope of the project very specific and clear. If the design and implementation are simple, testing becomes easier and effective. As in BRIDGE process model all the phases are distinctly well defined, it promotes simple design and implementation of the system that makes the other subsequent activities easier, simple and more effective.

- **Self-organizing Teams :** By self-organizing team we mean that the team members share a common goal and belief that their work is interdependent and collaboration is the best way to accomplish their goal. Rather than having a manager with responsibility for planning, managing and controlling the work, the team members share increasing responsibility for managing their own work and also share responsibility for problem solving and continuous improvement of their work processes. Hence, the empowered team members' reduce their dependency on top management as they accept accountability. The team structure places ownership and control close to the core of the work. The primary role of the project management is to build individual stakeholder a dedicated responsibility centre which is easy to establish through BRIDGE process model. By developing individual responsibility centers, the team may become the self-organizing team.

  Advantages of Self-Organizing :

  1. People in a self-organized team are able to make decisions themselves and accordingly adapt to changing situations.
  2. Self organized teams do a much better job of utilizing the talents of the team because more minds are involved in any activity.
  3. Self organized teams have much more communication between team members.

4. The best way to learn is to have actual responsibility and opportunities to do new things.
5. A self organized team is collectively aware of the upcoming work and much better able to bootstrap themselves with new work when they complete their existing task.
6. Self organized teams spread knowledge around much better and make decisions together. That makes each team member more effective because they have much more background on the "why" of the coding assignments.
7. A command and control team member often lacks an understanding of why a decision was made because they weren't involved with that decision. This may hamper their ability to follow a design or approach which restricts productivity.

As in BRIDGE model, the project management has consistent involvement with the development team, so if needed then the top management can facilitate the development team at any moment to be self-organized.

- **Internal Assessment for Knowledge Enhancement :** In association to project management the individual stakeholders may carryout internal assessment at regular intervals. Through the internal assessment result, the progress may be measured too. Further from the internal assessment the team may identify the various bottlenecks and upon rectifying and adjusting become more effective and plan the future activities efficiently. Being project management team with the development team in the BRIDGE process model, internal assessment becomes much easier.

- **Quality Assurance :** In BRIDGE model quality is ensures through implementation of multi level quality improvement methods through phase-wise verification, unit level, integration, and system testing, and system validation. Further, during maintenance, discovered errors if any may be rectified. Additionally, being management team working together with the development team, the entire process and individuals remain under proper control and monitoring of the management teams. Overall, the integrated project development environment of this process model ensures the system quality to achieved and improved.

- **Economic Development :** Economic development is achieved through optimal utilization of the resources and restricting misuse of resources. The optimal resource management is done through management authority with individual's care

and concern. In BRIDGE, all the stakeholders works together with better coordination and concern ensuring economic system development.

7.7 Conclusion

Despite of the criticism towards traditional software development process, the goodness of agile process is questionable. Agile may not be the good choice for some projects always. Often, traditional process model proves to the better than agile for some types of projects. Hence, our objective should not be to criticize the traditional process models over agile, rather we must carry out research to accommodate the good attributes of agile process in traditional process models. In this chapter we have shown that the philosophy of agile may also be achieved through the BRIDGE process model which follows the principle of traditional software development too. Hence, we recommend BRIDGE process model to be practiced by industries for modern software development projects.

Chapter 8

Identifying the Reasons for Software Project Failure and Some of their Proposed Remedial through BRIDGE Process Models

8.1 Introduction

Software project failures are one of the primary reasons for increased cost of software product and services. There are enough evidences of project failures in past and present. Any organizations have to compensate the cost of the failure projects from the success projects. For these reason, software are still beyond the scope of small and medium scale companies causing significant impact on both social and economical factors. Apart from this, starting from economic losses to live losses is also caused by software project failures. Hence, it is important to identify the different reasons for software project failures. If these reasons are pre-known, actions can be taken during project development to reduce project failure risks.

At the beginning we have discussed about the criterion to evaluate a software project to be called successful or failed. Then, we have identified, categorized and briefly discussed different the root causes of project failures based on their source areas. Next, we have briefly highlighted the primary features of the BRIDGE (62) process model and explored the various ways and means to reduced or alleviate these project failure reasons by following the BRIDGE process model.

Based on the publication in the "*International Journal of Computer Sciences and Engineering (IJCSE)*", E-ISSN: 2347-2693, 3(1), 118–126, 2015.

8.2 Research Goal and Objectives

The goal of this work is to identify the different reasons for software project failure and categorization of those reasons based on their originating sources. Further, we have tried to find out the project failure risks especially originating from software process model and to propose their remedial strategy specially through following the BRIDGE (62) process model.

8.3 Definition of Successful and Failed Software Projects

The primary objective of software engineering is to develop software that agreed upon functionality and:
a. within Time
b. within Budget, and
c. with Good Quality.
Any software development project that satisfies the above criteria is to be called successful. According to Keider (153) and Saleh (154), a project should deliver agreed upon functionality on time and within estimated budget. Successful software project maybe defined as any software project that is set to support initially approved functionality, as well as the project comfortably satisfying the stakeholders and being accepted and largely used by the end users after deployment. Hence, Software project failure is defined as any project that is set to support the operations of an organization by exploiting the resources of information technology that fails to deliver the intended output within the originally allocated cost, time schedule (155).

8.4 Project Failure Statistics

To highlight the importance of this study, in this section some statistical data about the software project failure are shared. The survey statistics about software project failure and project estimate overrun carried out by Standish Group International i.e. the CHAOS Manifesto (156), in 2013 are given in Table 8.1 and Table 8.2:
From the statistical data presented in Table 8.1, it is observed that the alternative year average of project successful rate is 30.3%, project challenged by 46% and project failed by 23.4%.
From the statistical data presented in Table 8.2, it is observed that the alternative year

Table 8.1: Project Performance Statistics (in%)

Year	Successful	Challenged	Failed
1994	16%	53 %	31%
1996	27%	33%	40%
1998	26%	46%	28%
2000	28%	49%	23%
2002	34%	51%	15%
2004	29%	53%	18%
2006	35%	46%	19%
2008	32%	44%	24%
2010	37%	42%	21%
2012	39%	43%	18%

Table 8.2: Project Estimates Overrun Statistics(in%)

Year	Time Overrun	Cost Overrun	Features Delivered
2004	84%	56%	64%
2006	72%	47%	68%
2008	79%	54%	67%
2010	71%	46%	74%
2012	74%	59%	69%

average of project time overrun rate is 76%, project cost overrun 52.5% and project feature delivery rate is 68.4%.
Rupinder Kaur and Dr. Jyotsna Sengupta in their paper (157) presented the following statistical data:

- As per the Research Report of ESSU (European Service Strategy Unit), 57% of contracts experienced cost overruns, 33% of contracts suffered major delays, 30% of contracts were terminated, and 12.5% of Strategic Service Delivery Partnerships have failed.
- As per the KPMG Survey, on average, about 70 % of all IT-related projects fail to meet their objectives.
- From the presentation on software failure by Bob Lawhorn following statistics are presented:
  - Poorly defined applications (miscommunication between business and IT) contribute to a 66% project failure rate, costing U.S. businesses at least $30 billion every year.

- 60%-–80% of project failures can be attributed directly to poor requirements gathering, analysis, and management.
- 50% are rolled back out of production
- 40% of problems are found by end users
- 25%— 40% of all spending on projects is wasted as a result of re-work.
- Up to 80% of budgets are consumed fixing self-inflicted problems (Dynamic Markets Limited 2007 Study)

Research indicates that more than 50% of all IT projects become runaways–overshooting their budgets and timetables while failing to deliver the expected outcomes (155, 158). Johnson (155) reported that the overall project success had increased from 16% in 1994 to 28% in 2000. That makes it very curious, but probably not surprising, that according to an article in the IEEE Spectrum, about 10% of projects are abandoned either before or after completion, because the end product will not actually resolve the original business challenge (159).

8.5 Common Reasons for Software Project Failure and their Categorization

Often it is easy to identify whether a software project is successful or failed. But, it is really a tough job to identify and understand the actual reasons for project failure. For example, if the delivered system fails to meet the needs of the customer or user, the first question to ask is, "Why?":
- Was it because the development group didn't do a good job? Or
- Perhaps the requirements were not properly gathered or used? Or
- May be the people responsible for supplying the requirements were inaccurate? Or
- Was it something else?

Further, being software development a people intensive job, it is more complex to identify the exact reason to failure and to provide solutions to project failure. Usually, often there are multiple factors causing a software project to fail.

Possible areas/sources of Project Failure Reasons:

Form the above discussions it is easy to understand that there are several possible areas or sources of reasons to project failure. Some of the investigated sources of causes to software project failure are explored and listed below:

- People Sources
- Technology Sources
- Process Sources
- Organizational Sources
- Management Sources
- Business Sources
- Project Sources

Some reasons for project failure are easy to classify as belonging to one area or another, but some are harder to categorize even. So far significant effort has been made to identify and analyze the causes of software project failure discussed below (155, 157, 159, 160, 161, 162, 163, 164, 165). Now we try to identify the possible project failure reasons from the different sources as identified above.

A. Project Failure Reasons Originating from People Sources

In a software development project typically three types of stakeholders are associated:

- **Users:** Some of the project failure causes originating by these types of people may be due to:
  - Poor User Input
  - Lack of User Training
- **Client:** Some of the project failure causes originating by these types of people may be due to:
  - Conflicts
  - Politics
- **Project Development and Management Team:** Some of the project failure causes originating by these types of people may be due to:
  - Poor Quality Work by developers
  - Poor Quality Work by Management Personals

The project failure reasons may originate from one or more of these people sources. *Firstly*, often, the users are either unable to deliver the exact requirements to be delivered by the system or even may not be clear during initial stages of the development. As a result the project may fail due to wrong or inadequate user input. *Secondly*, often the project client and system users are different. Because of poor or misunderstanding, lack of communication gap between them, the client may convey wrong information and requirement to project development team that may lead to project failure. *Thirdly*, the project may fail because of the development team itself. These days, software development teams have become distributed in nature. The lack of communication among the development team and inefficient human resources may become the bottle neck for the project success. *Finally*, insufficient and inefficient project management team may lack to provide necessary management support to the project causing project to fail.

B. Project Failure Reasons Originating from Technology Sources

The rapid technological advancements are often good, but not always. Being software development a time intensive job, very often the technology used for the project implementation becomes obsolete before completion of the project causing the project to fail. Generally, the projects get canceled before their completion. Further, the technology used if not chosen wrong, may be new and immature failing to perform as expected causing project failure. From technology sources SW project failure may arise due to the following reasons:

- Wrong technology selection.
- Technology too new or didn't work as expected
- Use of immature technology
- Technology planning

C. Project Failure Reasons Originating from Process Source

Process failure is the largest and potentially the most pernicious of all sources of project failure and has been at the root of problems for decades. If the goal of a process is to produce a specific outcome, then anything that either delays or prevents the achievement of that specific outcome is a form of process failure. The process might deliver something, but if it does not deliver anticipated outcomes or does not meet expectations; the result is a failed process. This form of failure usually leads to finger pointing between development

groups and users, with each claiming the other did not understand (159). The root causes of SW project failure originating from process sources may include the following:

- **Wrong Process Selection:** There are many process models, but all have their own features and limitations. Often not all process models are suitable for any kind of projects. Thus process selection is typically challenging for any project implementation. Wrong or inappropriate process selection may lead to project fail.
- **Lack of User Involvement:** Non involvement of user and customers in the development process is one of the principle reasons that software does not fully meet customer expectations.
- **Lack of Communications:** When we think about communications failure the first thing that comes to mind is, "It's their problem,"and it is usually an internal dialog. However, lack of communications with end user or customers is rarely immediately considered, and it turns out to be one of the major problems. Further, delayed communications or communications latency is blamed as the reason for failure: "They didn't get back to me in time."
- **Unnecessary Processes:** Unnecessary processes apply to wasted or duplicated effort as well as a management or reporting structure that adds "heavy-weight" reporting and accountability to the development process.
- **Careless, sloppy, or missing software development processes:** Sloppy development process is the core value for the software engineering movement, contributed to the acceptance of object-oriented programming, helped fuel the agile movement, and more. Consider the customer at every step in the development process. While that will not guarantee the development process will be free from sloppiness, it will help focus on what is important.
- **Non-adaptability of process to Changes:** More importantly, the presence of one or more of these process failures contribute to business failure if the organization is not able to respond to changing business or market conditions. They also make it difficult to respond to customer perceived incidents that disrupt service delivery.

D. Project Failure Reasons Originating from Organizational Sources

- New to business– lack or no prior experience

- Improper Organizational Structures in respect to project need
- Poor communication among customers, developers, and users
- Reasons Related to Human Resource
- Insufficient resources
- Organizational culture and structure

E. Project Failure Reasons Originating from Management Sources

Additional project failure reasons may originate from project management sources. Some of the identified reasons contributing to project failure in this respect are as follows:

- Poor communication among customers, developers and users with management
- Lack of leadership and effective management
- Poor reporting of the project's status
- Insufficient involvement of senior management
- Insufficient staff/team size
- Inaccurate estimates of needed resources
- Lack of proper project management and control
- Sloppy development practices
- Failure to plan
- Commitment and patterns of belief
- Poor quality management and control

F. Project Failure Reasons Originating from Business Sources

In this section we focus on different causes of failures at the business level (159) that directly affect a software development project:

- **Non adaptive to changing conditions:** One of the most obvious forms of business failure also turns out to be the primary reason that development organizations cannot readily adapt to changing conditions: specifically, lack of management commitment.

- **Poor selection and use of a particular tool or vendor:** Another potential source of business failure is the management requirement that dictates the use of a particular tool or vendor without considering the outcomes expected by customers.
- **Commercial pressures:** Often there is commercial pressure on the project from business sources. Time-to-market, competition in business, economic breakdown, economic competency among similar products are the different sources of commercial pressures.

G. Project Failure Reasons Originating from Project Sources

Different projects are of varying nature, types and complexity. Often, there are many intrinsic reasons to the project itself causing the project to fail! These reasons may be related to the system requirements, risks, budget, schedule etc. Some of the project related reasons originating from the project itself are identified and listed below:

- Reasons Originating from System Requirements
    1. Lack of proper understanding and poor definition of system requirements
    2. Changing system requirements and project scope
    3. No more need for the system to be developed
- Reasons Related to Project Risk
    1. Poor project risk identification, management and control
    2. Late project failure warning signals
    3. Unrealistic or unarticulated project objectives and goals
- Reasons Related to System
    1. Project's complexity
    2. Poor system architecture and specification
    3. Critical quality problems with software
- Reasons Related to Budget and Schedule
    1. Inaccurate/over budgeting
    2. Hidden costs of going "Lean and Mean"
    3. Unrealistic and over schedule estimation

The Avanade Research Report (2007) (157) disclosed that 66% of failure due to system specification, 51% due requirement understanding, and 49% due to technology selection. Further, TCS (Tata Consultancy Services) 2007 (157) reported that 62% of organizations experienced IT projects that failed to meet schedules, 49% suffered from budget overruns, 47% had higher than expected maintenance costs, 41% failed to deliver the expected business value and ROI, 33% file to perform against expectations.
For detailed discussions about the BRIDGE (62) process model and its primary properties one may refer to Chapter 6 of this thesis and for the schematic diagram of the BRIDGE process model one may refer to Figure 6.2.

8.6 Remedial to Project Failure Risks through BRIDGE Process Model

As we have discussed in the earlier sections, the project failure reasons may originate from different sources of the project i.e. people, technology, process, organizational, management business and project sources. It is not possible to address all these project failure reasons only by following any process model. However, many of these failure reasons directly or indirectly related to the software development process model. So by following any suitable process model, many of this project risk can be reduced.
In this section we discussed the remedial(166) to some of the project failure reasons identified in the earlier sections by following BRIDGE (62)process model.

A. Remedy to Project Failure Reasons Originating from People Sources

- **Poor User Input:** In most of the process models, users are involved only during the initial phases of the development process when often the system requirements are unclear and ambiguous. Hence, the user inputs are often incorrect and poor. As the development proceeds, the requirements start getting clear. But by these times, the users are not a part of the development process. Hence, user input remains poor causing risk to project success. As in BRIDGE, the user are involved over the entire development process, the user remains the scope to provide update inputs that increases the rate of project success.

- **Lack of User Training:** Many of the organizations do think that only development and delivery of the system is the only responsibility of them. Thus they

don't often take user training as serious part or their interest. But the truth is that if the users are unable to use the system easily and efficiently, the project fails irrespective of how good the developed system may be! Proper and good documentations are the key to user training but are often ignored by many organizations. In BRIDGE, special focus is given on documentations at different phases of the development process. Thus simultaneously at the end of all phases proper documents are produced that may help during the user training process and self learning.

- **Client Conflicts and Politics:** Both of these issues arise because of the lack of user involvement. The scope of these problems may be reduces only by involving the users in the development process making the user themselves to be an individual responsibility centers in the project, which is supported in BRIDGE process model.
- **Poor-Quality Work by developers and Management Personals:** There are two possible reasons for this problem to arise:
  - Lack of Knowledge, Skill and Expertise of Developers: In this case, the process model doesn't have any role to play; rather it is an organizational staffing and human resource related problem to be managed at organizational level.
  - Because of Lack of Tendency to Quality Work of Developers: However, this reason may be handled by proper monitoring and management control efficiently by following BRIDGE as project management team is always with the development team.

B. Remedy to Project Failure Reasons Originating from Process Sources

- **Remedial to Wrong Process Selection:** The basic reasons for selecting wrong process model are unclear process objectives and goals. Further, as the different features of any process models are not distinct and ambiguous, people often selects wrong process model. In BRIDGE, the feature of the process model is very clear and unambiguous; the concerned may judge the suitability of this model for any typical project easily.
- **Remedy to Lack of User Involvement:** One of the primary features of the BRIDGE process model is the involvement of client over the entire development process that alleviates the project failure risks.

- **Lack of Communications:** This problem may also be originated from management sources. Often in no process model except BRIDGE all the stakeholders including project management team works together. Working together by different project stakeholder in BRIDGE, the communication gap is reduced among them.

C. Remedy to Project Failure Reasons Originating from Management Sources

Remedial support to project failure causes originated from management sources are beyond the scope of any process model. For example, the quality and expertise level of the individual management and development personals do not comes under the scope of development process. Thus risks i.e. lack of leadership and effective management, poor reporting of the project status, insufficient involvement of senior management, insufficient staff/team size, inaccurate estimates of needed resources, lack of proper project management and control, sloppy development practices, failure to plan, commitment and patterns of belief, poor quality management and control etc. depends on the quality and effective management team.

However, given an effective project management team, due to lack of direct involvement to the development process, some of the above problems may also arise, but in BRIDGE working all together under same umbrella automatically gets reduced.

D. Remedy to Project Failure Reasons Originating from Project Sources

- **Alleviating Reasons Originating from System Requirements Sources:** In BRIDGE, to alleviate reasons relate to lack of proper understanding and poor definition of system requirements, there is a dedicated phase for requirement gathering, analysis and specification that has to satisfy both the phase entry and phase exit criteria to get qualified. Further, being customer in BRIDGE always available to system analyst and developers there remains a scope to clear the doubts related to system requirements over the development process. It is also known that we may achieve the agile philosophy following BRIDGE process model (167). Thus accommodation and adaption of changing requirement becomes easy following this process model. Moreover, as BRIDGE process model promotes Component Based Software Development (CBSD) approach (115), changing project scope becomes easy by unplugging and plugging additional software components providing services as demanded. But in case of no more need for the system to be developed, no process model can help at all, as the case with BRIDGE.

- **Alleviating Reasons Related to Project Risk:** To alleviate risks related to risk identification, management and control, and late project failure warnings signals, in BRIDGE one phase is dedicated to feasibility study and risk analysis. Further, verification activity at the end of the individual phases helps to identify and reduce these types of project failure risks.

- **Reasons Related to System Complexity:** The well known tool and technique to manage project complexity is abstraction. Using software components and CBSD approach (167), BRIDGE has the quality to handle project complexity issues. In relation to poor system architecture and specification issues, to promote CBSD, in BRIDGE there is a distinct phase for architectural design of the system apart from detailed design and at the end of the all individual phases forceful specification is mandatory.

- **Related to Software Quality Assurance:** To ensure quality system development irrespective of human related issues, BRIDGE recommends to perform verification at the end of each development phases and to perform validation and testing of the system before system deployment. Further, to ensure quality development, the organizations additionally may follow the guidelines and recommendation given by different standard bodies i.e. SEI, ISO, and Six-Sigma etc. to attain different CMM levels, ISO certifications etc.

The remedy to other project failure reasons/risks originating from other difference sources i.e. technology, organization, business are beyond the scope of capability of any process model. Further, problem related to project budget and scheduling depends heavily on the degree of expertise level of the project manager/estimator and are beyond the capability scope of any process model as the case with BRIDGE.

8.7 Conclusion

In this chapter we have identified the different reasons contributing to project failure from there originating sources. Then we have highlighted the features of BRIDGE process model and discussed at length on how some of these project failure reasons may be reduces following BRIDGE process model. The comparative analysis of BRIDGE with some other well known process model explored the distinguished features of BRIDGE over other (168, 169). Thus, we conclude by recommending the BRIDGE process model to be followed for SW development projects to gain project success rate.

Chapter 9

A Comparative Analysis of BRIDGE and Some Other Well Known Software Development Life Cycle Models

9.1 Introduction

The rapid development in the hardware technology has made modern processors very efficient and powerful. Hence, the expectations from the software have gone to zenith. But the complexity of the modern software are much complex as compare to those of earlier. Development cost, time and quality of the modern software are in crisis. A software (SW) project, irrespective of its size, goes through certain defined stages, which together, are known as the Software Development Life Cycle (SDLC). Life Cycle refers to the different phases involved starting from the project initiation to project retirement. For better understanding and implementation of the various phases of software development, different software development models have been developed and proposed so far. There are several Software Development Life Cycle (SDLC) Models i.e. Classical Waterfall, Spiral, Prototype, V-Model, evolutionary model etc. All these SDLC models have several advantages as well as some limitations. It is pre established that different SDLC models have different capabilities and limitations. Hence, selecting suitable SDLC model for any project is quite crucial as not all process models are good for any type of project. Hence, analyzing the different SDLC model is significant and helps one to select the appropriate model for a project. Recently a few more new process models are

Based on the publication in the "*International Journal of Computer Science & Engineering Technology(IJCSET)*", ISSN: 2229–3345, 5(3), 196–202, 2014.

proposed with the well known traditional models to accommodate the new industrial needs.

9.2 Software Development Approach, Process and Process Model

It is really tough to draw a sharp line between software development approaches and SDLC process models. In many literature of software engineering, these terms are used interchangeably or confusedly. So, before we begin the details discussion of the topic, let us somehow draw the boundary line between software development approaches and SDLC process models. Defining these two terms are beyond the scope in this context. Here we just try to explain both only to establish the differences from our point of view. SW development process or simply process typically defines the set steps to be carried out during the development of the system. SW development life cycle (SDLC) is the time from the concept development to the product retirement i.e. the time of SW process. SW development life cycle (SDLC) process model typically depicts the fashions in which the SW process to be carried out i.e. which steps/phases to be done before or after another step/phase. In general all the process models do cover all distinct phases defined by SW process, but in different manner or sequence- which makes one process model differ from the other. In other words, a software development process model is an approach to the Software Development Life Cycle (SDLC) that describes the sequence of steps to be followed while developing software projects (7, 8). We consider Agile, incremental, extreme and iterative as approach or philosophy to software development rather than as process model which can be implemented following other process models i.e. Waterfall, RAD, Spiral, Prototyping or alike.

9.3 Parameter Selection for Comparative Analysis

Below we enlist and discuss briefly the parameters alphabetically those we have considered for the purpose of comparative analysis:

- **Adaptability:** This is the ability to react to operational changes as the project is developed. Change orders are easily assimilated without undue project delay and cost increases.

- **Budget:** Budget remains one of the most significant crisis for software development projects. Some process model like Spiral and Prototyping increases the project cost as compared to others. Hence a SDLC process model has great impact on software development cost or budget.

- **Changes Incorporation:** Change is unavoidable in software development. Managing change is a critical component of any SDLC model. Change Management and SLDC are not mutually exclusive. Change management occurs throughout the development life cycles which need to be incorporated in the system development.

- **Complexity of the Process Model:** Different SDLC process model have varying degree of complexity. Some are easy to use and implement while others are not.

- **Documentation:** Documentation of software development process is very important but time consuming and expensive. To reduce development time and cost, agile philosophy recommends less document which remains one of the most important critic of agile philosophy. Documentation plays vital roles in system development, implementation, maintenance, and project management. But, not all process models facilitate and recommend adequate and sufficient documentation.

- **Expertise Required:** Although some process models are better over others, but need some kind of expertise during its use and implementations in various phases at varying degrees. To avail the advantages of some process models i.e. Spiral, BRIDGE(62) and others which supports reusability - the software engineers required certain level of expertise.

- **Flexibility:** The freedom afforded to software architects, analysts or developers to tailor the software development process according to business needs and project characteristics is a crucial factor in successful project completion. The software development organization often can benefit from introducing flexibility into their software development methodologies (170).

- **Guarantee of Success:** This is really crucial to measure whether any process model will guarantee success or not. If so, up to what extent wills the process model guarantees the success is a big question need to be explored. As the project success depends upon many other constraints and parameters, but given the other parameters as desired, project success may vary from following one SDLC model to another.

- **Integrity & Security:** Including security early in the system development life cycle (SDLC) will usually result in less expensive and more effective security than adding it to an operational system. To be most effective, information security must be integrated into the SDLC from its inception (171).

- **Maintenance:** Systems are dynamic and the model offers the ability to produce a final project that is inherently designed for maintenance. This includes such items as cumulative documentation.

- **Management Control:** Management will have the ability to redirect and if necessary redefine the project once it is begun. A key phrase is 'incremental management control', with each step under tight management control. Management control has great impact on project success.

- **Overlapping Phases:** Each step of the project is to be completed before another is begun. Project modules are distinct and easily identifiable.

- **Parallel development:** Parallel development support, if possible to employ may increase productivity and reduce development time while optimally utilizing the resources.

- **Productivity:** The SDLC must ensure that the expected return on investment (ROI) for each project is well defined. The SDLC must minimize the unnecessary rework. It must be designed in such a way as to take maximum advantage of the computer assisted software engineering (CASE) tools (119). At the same time the SDLC must utilize the resources most effectively and efficiently to improve the productivity.

- **Progress Measurement:** Progress measurement allows development team as well as the project management team to determine how well tasks were estimated, how well they were defined, and whether items are completed on-time and within budget. Any SDLC process model should provide the facility to measure the progress during the system development.

- **Quality Control:** Each module of the project can be thoroughly tested before another module is begun. Project requirements are measured against actual results. Milestones and deliverables can be used for each step of the project.

- **Requirements Specification:** Depending on the project nature, the requirement may be identified at the very beginning of the project development or may be discovered during the development process. But, not all process model supports requirement discovery over the development process. Hence, requirement specification may be static or may be dynamic. Any SDLC process model should take into account the issue of requirement specification.

- **Requirements Understanding:** Some process model needs the requirement must be well understood before the development process stated, while other may allow understanding the requirements over the development process. One may start with the initial understanding of the requirement and during the development the requirement understanding increases gradually.

- **Reusability:** Reusability is one of the most significant and efficient attribute of any SDLC process model these days. Reusability helps to improve system productivity, reduce cost and system delivery on-time. The degree of reusability support may vary from one process model to another.

- **Risk Involvement:** The risk involvement may vary from one model to another depending on the nature of requirement understanding capability support by the process model. Apart from this, there may be several other sources of risks involvement.

- **Risk Management:** Different types of risks are implicit part of any project. Levels of risk are identifiable and assessment strategies available. Strategies are proved for overall and unit risks.

- **Simplicity:** Any model need is easy to understand and to implement. Simplicity of any process model reduces the burden of expertise and improves productivity while reduce development cost and project risk.

- **System Delivery:** The system may be delivered either partially as individual operational module wise or as the complete system with full functionality at once.

- **Time:** Time is actually referred to as Time Horizon because we are interested in knowing the projected completion of the project. The development time may vary from one process to another.

- **Understandability and Implementation:** Different process model may need varying level of expertise. Simple and better understandable process model are always easy to implement.
- **User Involvement:** Any model lends itself to strong and constant end user involvement. This includes project design as well as interaction during all phases of project development.

9.4 Comparative Analysis

The different process models are discussed in section 2.5 and BRIDGE process model is discussed earlier in Chapter 6. The comparisons among different SDLC models in respect to the features discussed in the previous section are illustrated in Table 9.1 (105, 172, 173, 174, 175, 176, 177, 178, 179, 180). From the above comparative analysis, it is established that the BRIDGE process model possesses many suitable features in comparison to the other process model.

9.5 Conclusion

There exist several well known SDLC process models. One process model has different comparative advantages from the others in many respects. But no process model is just good for any type of project. So it is not blindly recommended to choose any process model for any project! The above comparative study shows that overall the BRIDGE process model has several competitive advantages over the other existing well known process models. As BRIDGE model has excellent adaptability, supports process tailoring and other attributes, we recommend this SDLC process model to be used for any types of software development projects.

Table 9.1: Comparison of Different SDLC Process Model

Models / Parameters	BRIDGE	Waterfall	Prototype	Evolutionary	Spiral	RAD	V-Shape
Adaptable	Excellent	Limited	Good	Good	Excellent	Limited	Limited
Budget	Low	Low	High	Low	High	Low	Low
Changes Incorporate	Easy	Impossible / Difficult	Easy	Easy	Easy	Easy	Difficult
Complexity	Medium	Simple	Moderate	Complex	Complex	Medium	Simple
Documentation	Yes	Strong	Weak	Moderate	Moderate	Poor/ Limited	Yes
Expertise Required	Medium	Low	Low	Low	High	Medium	Medium
Flexibility	Flexible	Inflexible	Highly Flexible	Highly Flexible	Flexible	High	Rigid
Guarantee of Success	High	Less	Good	Good	High	Good	High
Integrity and Security	High	Vital	Weak	Weak	High	Vital	Limited
Maintenance	Easily Maintained	Least Glamorous	Routine Maintenance	May be overlooked	Typical	Easily maintained	Lest
Management Control	Yes, Dedicated	No	No	Weak	Moderate	Weak	Weak
Overlapping Phases	May be	No	Yes	Yes	Yes	No	No
Parallel Development	Supported	No	No	Limited	Limited	No	Limited
Productivity	Highest	High	Improved	Improved	High	Improved	Improved
Progress Measurement	Measurable	Easily Monitored	Measurable	Measurable	Measurable	Measurable	Measurable
Quality Control	Very Good	Poor	Moderate	Good	Good	Adequate	Moderate
Requirements Specification	Adaptable/ Dynamic	At the Beginning	Frequently Changed	Frequently Changed	At the Beginning	Time-box Release	At the Beginning
Reusability	Excellent	Limited	Poor	Poor	Moderate	Moderate	Moderate
Risk Involvement	Low	High	Low	Moderate	Low	Little	Low
Risk Management	Highly Supporter	Not Considered	Moderate	Good	Highly Supporter	Poor	No
Simplicity	Intermediate	Simple	Simple	Intermediate	Intermediate	Very Simple	Simple
System Delivery	Early and periodic partial operational system	At the end of the system development	At the end of the system development	Early and periodic partial operational system	At the end of the system development	At the end of the system development	At the end of the system development
Time	Shortest	Short	Long	Long	Long	Short	Short
Understandability and Implementation	Moderate	Easy	Easy	Moderate	Complex	Moderate	Easy
User Involvement	Throughout Process	At the beginning	High/Up to design phases	Throughout Process	High	Throughout Process	At the beginning

Chapter 10

SRS BUILDER 1.0: An Upper Type CASE Tool For Requirement Specification

10.1 Introduction

Although, hardware costs are decreasing drastically, still the computers are not used extensively by the business organization because of the huge software cost. In recent years, Computer Aided Software Engineering (CASE) tools have emerged as one of the most important innovations in software development to manage the complexity of software development projects reducing its product cost. Using CASE tools over their SDLC process may reduce the developing cost significantly.

Although, almost all develop countries do use certain CASE tools, but its extensive use is still a dream because of their product cost. Although, big software giant's do use their own developed or commercially available CASE tools in development software, but their quality, applicability, cost, availability remains as big question. The huge cost of such commercially available CASE tools made these outreach of the small software development companies. In this study, we are going to demonstrate newly developed requirement specification tool name SRS BUILDER version 1.0.

The rest of the paper is organized as follows: We started by defining software engineering, Computer Aided Software Engineering and CASE tools. Then, we have laid down the different types of CASE tools and advantage of using CASE tools with its limitation. In the later section we have discussed about SRS BUILDER 1.0 with its sample design.

Based on the publication in the "*Proc. of The* 4^{th} *National Conference on Computing for National Development(INDIACOM-2010)*", February 25–26, 2010, ISSN: 0973–7529, ISBN: 978-81-904526-9-4

10.2 Computer Aided Software Engineering (CSAE) and CASE Tools

In the following we are going to define some terminologies used in software engineering.
Software Engineering (SE): Before defining CASE, reminding the definition of software engineering is justifiable. Although, the age of Software Engineering is quite old, but prior to its born, people do used to develop software. But because of some problems (discussion of these problems are beyond the scope of paper) faced later with those systems, software engineering emerged as a new subject.
The IEEE defines software engineering as (19) :
(i) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
(ii) The study of approaches as in (i).
More significantly, software engineering may be defined as the systematic approach to develop quality software within both budget and time constraints.
Computer Aided Software Engineering (CASE): CASE is an acronym that stands for Computer Aided Software Engineering. Roughly, this is all about using computers at different phases of the SDLC process during the development and maintenance of software to assistance the development team. CASE provides the software engineer with the ability to automate manual activities and to improve engineering associated to software development. Basically, it is all about using software to develop software.
CASE tools: CASE tools are the set of tools that permit collaborative software development and maintenance. These tools are concerned with automated tools that aid in the definition and implementation of software systems.
Formal definition of CASE:

- *Individual tools to aid the software developer or project manager during one or more phases of software development (or maintenance).*
- *A combination of software tools and structured development methodologies.*

Types of CASE tools: Depending on the activities performed, CASE tools are primarily divided in to three categories. They are as follows:

1. **Upper CASE tools:** Primarily focuses on the System analysis and design phase of the SDLC.

2. **Lower CASE tools:** Focuses on system implementation phase of SDLC.
3. **Integrated CASE tools:** It helps in providing linkages between the lower and Upper CASE tools.

10.3 Advantages of Using CASE Tools

With the growing importance of CASE tools, more steps in the SDLC are being automated. However, a complete automated software facility for all steps is still a dream. Presently available CASE tools cover only a certain modules of the general CASE facility. The benefits of using CASE tools are as follows:

1. Increased Productivity.
2. Product Quality improvement.
3. Development Cost Reduction.
4. Effort Reduction: Different studies carried out to measure the impact of CASE put the effort reduction about 30–40% (126).
5. Reduction in Development Time.
6. Reduce the drudgery and working style in a software engineer's work.
7. Create good quality documentation. Our, proposed tool basically focuses on this particular aspect of computer aided software engineering.
8. Create maintainable system.
9. Providing a uniform platform for software/system developers to present information and knowledge compactly for ease of communication (181, 182, 183).

10.4 Usage of CASE Tools: A Research Report

CASE (Computer Aided Software Engineering) tools are supposed to increase productivity, improve the software product quality and make Information Systems development a more enjoyable task (184). However, they have been failing to deliver the benefits they promise (185). The results of the research carried out by Kemerer (186) regarding the usage of CASE tools after introduction are shown in the Figure 10.1.

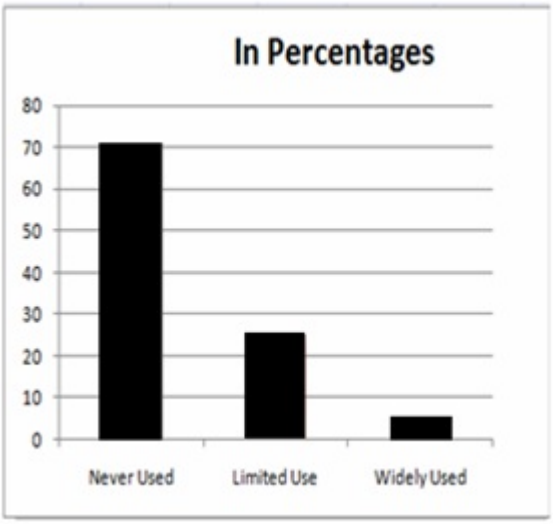


Figure 10.1: Use of CASE Tools in Organizations (in %)

This results about CASE tool usage raise a significant question, why the percentage of widely used case tools are too low i.e. 5% only?
The answer to the question is the limitations with the CASE tools discussed in the next section.

10.5 Limitation of CASE Tools

CASE tools are still in limited use because of the following limitations:

1. CASE tools don't support automatic development of functionally relevant system.
2. It force system analyst to follow a prescribed methodology.
3. It may change the system analysis and design process.
4. Poorly supported expected functionality from them.
5. Huge cost of the CASE tools.
6. Lack of Concern in CASE tool usage.
7. Bad quality of the CASE tools.

8. CASE tools are complex because they offer a large array of options and support for software development activities.

9. Cheap Labor cost: In developing countries like India and other underdeveloped countries, cheaply available human resources remain as one of the biggest reasons for not using the costly CASE tools.

10.6 Motivation for Undertaking the Research Project

In addition to usage in industry by practitioners, CASE tools are employed by educators to teach students software development skills. Evaluating and selecting a CASE tool for a specific systems development course is quite difficult.
While teaching the paper software engineering at university (NBU), it was found that the students are very much confused about CASE tools. The significant reasons for the same are basically widely available complex poor quality CASE tools which are quite expensive to purchase although. Some modules are good in some if not in all. The students do not feel interesting to use them. Then we under took the project to develop a new customized CASE tool for requirement specification supporting IEEE specification as per the students need that will help the student to improve their skill and have a clear vision about it. We start with the development of CASE tool to specify the system requirements to generate the System Requirement Specification (SRS) document. The outcome of our attempt is the so named, *SRS BUILDER 1.0*– the CASE tools to generate the SRS document. Moreover, we are planning to distribute the newly developed CASE tool to the educational institutions on demand almost free of cost with a nominal transportation cost for the sake of student community.

10.7 SRS BUILDER 1.0: The New Requirement Specification Tool

As from the SLC models, we know that after the requirements are gathered by the system analyst, the analyzed and finalized requirements need to be specify in the SRS document. The SRS document is then provided to the software development team for next phase of the development.

- **SDLC Model Followed**
  The BRIDGE software development process model (62) has been followed to develop the SRS BUILDER 1.0.

- **Product Quality Features**
  - Support to IEEE specification for SRS writing format along with the flexible other customized specifications as per your organizational need.
  - Good Graphical User Interface
  - Easy to use
  - Easy Installation
  - Incorporated User Documentation
  - Easily Available
  - Validated
- **System Specifications**
  - Front End: Visual Basic 6.0
  - Back End: MySQL
  - Operating System: Windows XP, Windows Vista.
  - Memory Requirement: 15 KB

10.8 Function Hierarchy Diagram of SRS BUILDER 1.0

The Function Hierarchy Diagram (FHD) of the SRS BUILDER 1.0 is shown below in Figure 10. 2:

10.9 Example: Sample SRS Organization Generated by SRS BUILDER 1.0

A typical SRS generated by the SRS BUILDER 1.0 for the ATM system is shown below in Figure 10.3 to Figure 10.6. This is not a complete and correct SRS for the intended system, but a sample used only to show the SRS organization generated by the tool. The font size and spacing has been changed to accommodate in the paper keeping the context organization unchanged.

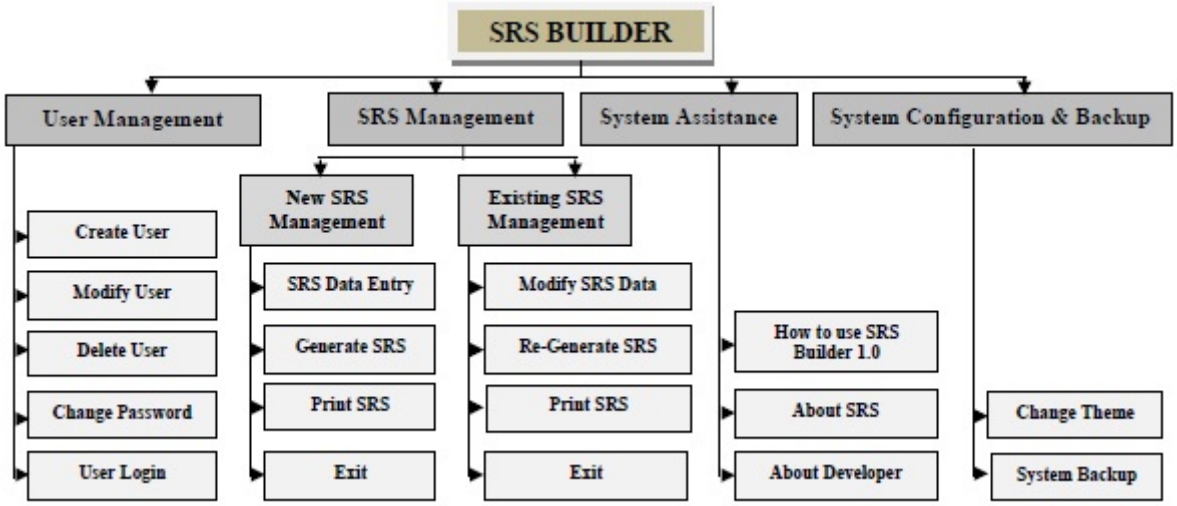


Figure 10.2: FHD (Function Hierarchy Diagram) of SRS Builder 1.0

Project Title: SRS ATM

Project Id: 1

SOFTWARE REQUIREMENT SPECIFICATION

Introduction

Purpose:

This document describes the software require……

Scope:

The software supports a computerized banking …….

Definition:

• Account:

A single account at a bank against which transaction……..

Intended Audience:

The intended audience of this SRS consists of:

• Software designers….

Reference: NA

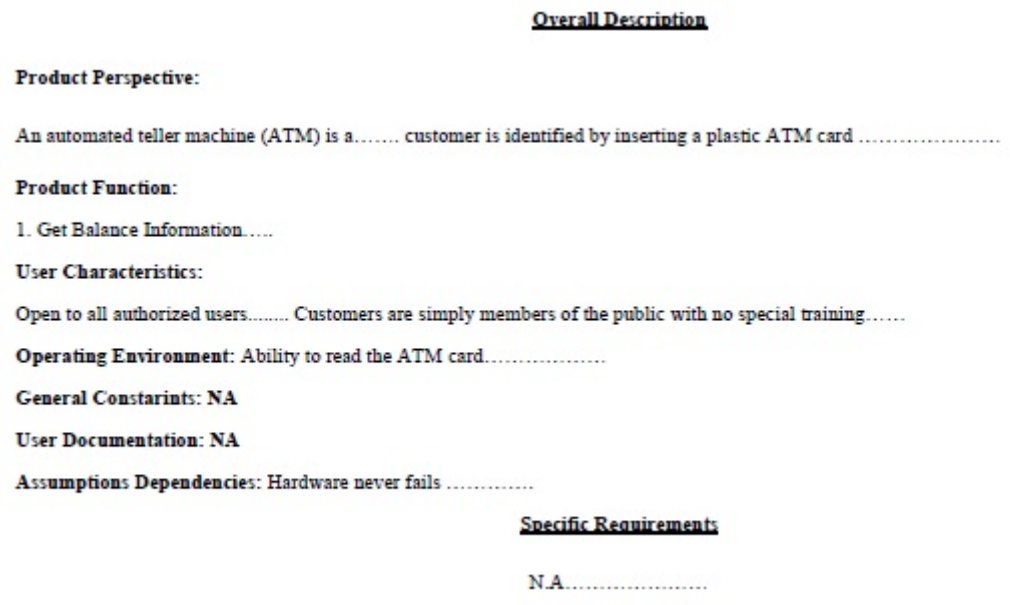
Overview: NA

Document Conventions: NA

Figure 10.3: A sample SRS Structure Generated Using SRS Builder 1.0 (to be continued...)

10.10 Conclusion

We may conclude by pointing out that, this CASE tool will play an important role to the software developers and learners to use and understand the utility of the CASE tool in



Overall Description

Product Perspective:

An automated teller machine (ATM) is a……. customer is identified by inserting a plastic ATM card ……

Product Function:

1. Get Balance Information…..

User Characteristics:

Open to all authorized users..... Customers are simply members of the public with no special training……

Operating Environment: Ability to read the ATM card……

General Constarints: NA

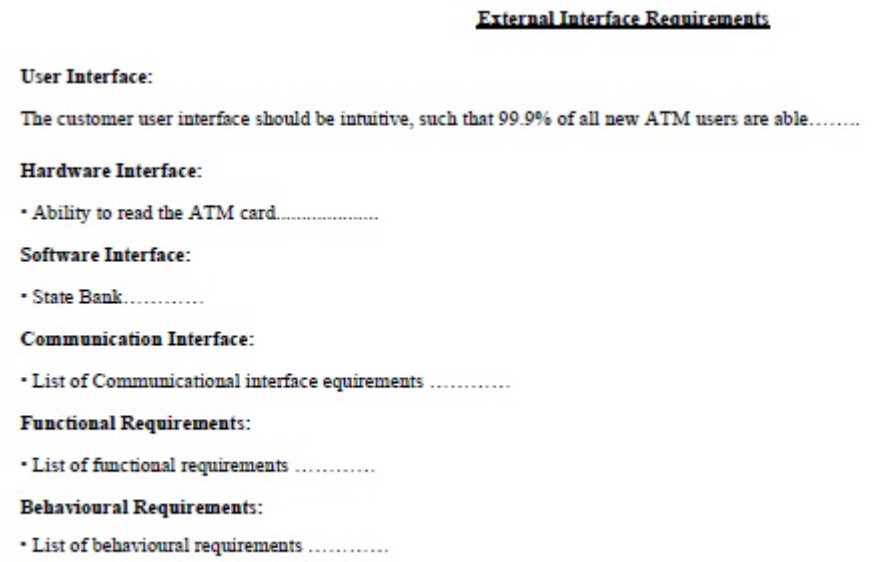
User Documentation: NA

Assumptions Dependencies: Hardware never fails ……

Specific Requirements

N.A……

Figure 10.4: A sample SRS Generated Using SRS Builder 1.0 (to be contd...)



External Interface Requirements

User Interface:

The customer user interface should be intuitive, such that 99.9% of all new ATM users are able……..

Hardware Interface:

• Ability to read the ATM card……

Software Interface:

• State Bank……

Communication Interface:

• List of Communicational interface equirements ……

Functional Requirements:

• List of functional requirements ……

Behavioural Requirements:

• List of behavioural requirements ……

Figure 10.5: A sample SRS Generated Using SRS Builder 1.0 (to be contd...)

todays'complex software projects. Also, as we mentioned earlier, interested educational institutions and organizations may contact the author for the CASE tool for their usage.

Other Non-functional Requirements

Performance Requirements:

• It must be able to perform in adverse conditions like high/low temperature etc. .....

Safety Requirements:

• Must be safe kept in physical aspects, say in a cabin .....

Security Requirements:

• Users accessibility is censured in all the ways .....

Software Quality: NA

Other Requirements: NA

SYSTEM REQUIREMENTS SPECIFICATION for ATM Withdrawal

Submitted by:

Program Manager/Functional Project Officer	Date

Coordination:

Director, Applications Architecture	Date
Director, Engineering	Date
Test Director	Date

Approved by:

Functional Manager	Date

Figure 10.6: A sample SRS Generated Using SRS Builder 1.0

Chapter 11

Summary and Conclusion

11.1 Summary of Work

History of any subject let us know the archived knowledge of the subject and its matter. Hence, at the beginning to know and understand the origin and its significance we have presented the history of software engineering in the chronological order of evolution and some modern trends in this field.

As this work is specifically focused on the Software Development Life Cycle (SDLC) Model, some of the existing and well known software development models with their origin, advantages, limitations and suitability from industrial perspectives are also presented.

We discussed the general methodologies used for software engineering research. Here, we have also mentioned and justified that our work is based on the applied research methodology where we undertook to solve some issues related to software crisis following a typical process model– named BRIDGE (62).

We have also explored the importance and contribution of Component Based Software Development (CBSD) as a tool to alleviate software crisis.

The different desired characteristics of any process model are also investigated, identified and presented.

Then, the BRIDGE life cycle process model is presented. Agile Software development philosophy is relatively young in this era. Some principles of agile is contradictory with the traditional development principles. Further, we have established that the agile philosophy may also be achieved following the BRIDGE process model.

Next, the different reasons attributing to software project failure are pointed out in this work with their remedial through BRIDGE process model .

The comparative analysis of the BRIDGE process model with some other existing models are made and presented.
A case study of the design and development of SRS BUILDER 1.0, a CASE tool following the BRIDGE process model is also presented.
We conclude by recommending the BRIDGE software development life cycle process model to be followed and practiced for any types of software development projects by different software development organizations to gain project success rate.

11.2 Concluding Remarks

An important initial step in addressing software problems is to treat the entire development process as a performable, controllable, measurable and improved process as a sequence of tasks that will produce the desired result. Any fully effective software process must consider the interrelationships of all the required tasks, tools and methods, skills, training and motivation of the people involved. At the same time, an ideal SDLC process implementation should be quicker, cost effective and easy to implement and follow. It should also be stakeholder and project team friendly. A process model should also address the top challenges experienced by project managers. These days, Component Based Software Development (CBSD) approach has been successful to provide significant contribution in alleviating the software crisis, but rarely any process model directly promotes this approach.
The primary objective behind developing BRIDGE process model was to consider the above issues and are addressed efficiently. It is shown that the philosophy of agile may also be achieved through the BRIDGE process model following the principle of traditional software development too. It is also observed that, many of the project failure reasons may be alleviated by following BRIDGE process model. This model have excellent adaptability, process tailoring support and also promotes the usage of CASE tools. The comparative study shows that the BRIDGE process model has several competitive advantages over the other existing well known process models. If the BRIDGE model is followed to any software development project, most of the software crisis may be overcome up to great extent delivering the fully functional system with better quality within time and budget achieving the true goal of any software project development.
It is concluded that the BRIDGE software development life cycle process model may be followed and practiced for any types of software development projects by different

software development organizations to gain project success rate.

11.3 Future Work

A number of necessary or potentially fruitful further areas of research were identified whilst conducting the work described in this thesis. Progress in these areas would enhance the quality and productivity of software development while decreasing the total product cost.

In this work we have identified and enlisted the desired characteristics from any process model irrespective of their degree of importance. In future, we shall investigate the significance and importance of the individual characteristics of the SDLC process model and shall prioritize them accordingly towards developing process metric. Following such process metrics, one may choose the suitable process model for any project which shall provide optimal solution and address other common issues related to process models.

We are implementing several instances of sample projects following BRIDGE and different other models individually by different teams to perform practical experimental comparative analysis. During the experiment we shall refine the BRIDGE model if necessary to make this model the best suitable for the industry usage. In near future we would also like to validate the result of the theoretical comparative analysis by means of practical experimental statistical results by collaborating with some industries to implementation and follow up its performances in real project development.

Bibliography

[1] Robert L. Glass. **Foundations of Software Engineering: An early history of software engineering**, January 1999. 1, 81

[2] A. Turing. **On Computable Numbers, with an Application to the Entscheidungsproblem**. In *Proc. London Mathematical Society*, pages 230–265, 1936. 3

[3] S. Kleene. **Recursive Predicates and Quantifiers**. *Transactions of the American Mathematical Society*, **53**(1):41–73, 1943. 3

[4] A. Turing. **Intelligent Machinery**. *Mechanical Intelligence 5*, **1**(1):3–23, 1969. 3

[5] B. W. Boehm. **A View of 20th and 21st Century Software Engineering**. In *Proc. of ICSE'06, May 20–28, 2006, Shanghai, China*, pages 12–29, 2006. 5, 6, 7, 8, 11

[6] Jr. Frederick P. Brooks. **Brooks, No Silver Bullet: Essence and Accidents of Software Engineering**. *IEEE Computer*, **20**(4):10–19, 1987. 8

[7] L. Guimares and P. Vilela. **Comparing Software Development Models Using CDM**. In *Proc. of The 6th Conference on Information Technology Education, New Jersey*, pages 339–347, 2005. 13, 141

[8] N. Ruparelia. **Software Development Lifecycle Models**. *ACM SIGSOFT Software Engineering Notes*, **35**(3):8–13, 2010. 13, 141

[9] W. Scacchi. *Encyclopedia of Software Engineering, 2 Volume Set*. John Wiley and Sons, second edition, 2001. 13

[10] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N. J., first edition, 1981. 13

[11] B. W. Boehm. **Software engineering economics**. *IEEE Transactions on Software Engineering*, **10**(1):4–21, 1984. 13

[12] B. Curtis, H. Krasner, and N. Iscoe. **A Field Study Of The Software Design Process For Large Systems**. *Communications Of The ACM*, **31**(11):1268–1287, 1988. 14

[13] W. A. Hosier. **Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming**. *IRE Trans. Engineering Management*, **EM**(8):99–115, 1961. 15

[14] W. W. Royce. **Managing the development of large software systems: concepts and techniques**. In *Proc. of the Ninth International Conference on Software Engineering*, pages 328–338, 1987. 15, 26

[15] L. Rising and Norman S. Janoff. **The SCRUM Software Development Process for Small Teams**. *IEEE Software*, **17**(4):26–32, 2000. 23

[16] A. Highsmith III James. *Adaptive Software Development*. Dorset House Publishing, first edition, 2000. 23

[17] M. Aoyama. **Agile Software Process and Its Experience**. In *Proc. of 1998 International Conference on Software Engineering*, pages 3–12, 1998. 24

[18] S. L. Pfleeger. *Software Engineering: Theory and Practices*. Pearson Education, second edition, 2007. 25, 85, 101

[19] R. S. Pressman. *Software Engineering, A practitioner's Approach*. McGrawHill International Edition, sixth edition, 2005. 25, 85, 93, 109, 115, 148

[20] I. Sommerville. *Software Engineering*. Pearson Education, seven edition, 1994. 25, 85, 93

[21] W. Winston Royce. **Managing the development of large software systems**. In *Proc. of IEEE Wescon, 1970*, pages 382–338, 1970. 26

[22] H. Gomma and D. B. H. Scott. **Prototyping as a tool in the specification of user requirements**. In *Proc. of Fifth Int. Conf. on Software Engineering*, pages 333–341, 1981. 31

[23] B. W. Boehm. **A Spiral Model of Software Development and Enhancement**. *IEEE Computer*, **21**(5):61–72, 1988. 36, 37

[24] J. Martin. *Rapid Application Development*. MacMillan, New York, first edition, 1991. 42, 44, 45

[25] P. Beynon-Davies, C. Carne, H. Mackay, and Tudhope. **Rapid application development (RAD): An Empirical Review**. *European Journal of Information Systems*, **8**(1):211–223, 1999. 42, 44

[26] P. Beynon-Davies, H. Mackay, R. Slack, and D. Tudhope. **Rapid Applications Development: the future for business systems development?** In *Proc. of BIT96 Conference, November 7th, Manchester Metropolitan University*, 1996. 44, 45

[27] E. Elliott. **Rapid Applications Development (RAD): an odyssey of information systems methods, tools and techniques**. In *4th Financial IS Conference, Sheffield Hallam University, U.K.*, 1997. 44, 45

[28] D. Tudhope, P. Beynon-Davies, H. Mackay, and R. Slack. **Time and representational devices in Rapid Application Development**. *Interacting with Computers*, **14**(4):447–466, 2001. 44

[29] C. S. Osborn. **SDLC, JAD and RAD: Finding the Right Hammer**. *Centre for Information Management Studies, Working Paper*, pages 95–107, 1995. 44

[30] P. Beynon-Davies, H. Mackay, and D. Tudhope. **It's lots of bits of paper and ticks and post-it notes and ..... a case study of a RAD project**. *Information Systems Journal*, **10**(3):195–216, 2000. 44

[31] P Beynon-Davies. **Rapid Applications Development (RAD)**. *Briefing Paper, Kane Thompson Centre, University of Glamorgan*, 1998. 45

[32] S. McConnell. *Rapid Development – Taming Wild Software Schedules*. Microsoft Press, Washington., first edition, 1996. 45

[33] J. Highsmith. *Agile Software Development Ecosystems*. Addison-Wesley, London, first edition, 2000. 45

[34] P. Kruchten. *The Rational Unified Process: An Introduction*. Pearson, third edition, 2003. 47, 50

[35] L. May Elaine and A. Zimmer Barbara. **The Evolutionary Development Model for Software**. *Hewlett-Packard Journal*, **August 1996**(Article 4):1–8, 1996. 50

[36] Rlewallen. **Software Development Life Cycle Models**, 2005. 50

[37] S. Redwine and et al. **DoD Related Software Technology Requirements, Practices, and Prospects for the Future**. *Institute for Defense Analysis, Alexandria, VA*, **P-1788**(1):357, 1984. 52, 60, 63, 64

[38] S. Redwine and W. Riddle. **Software technology maturation**. In *Proc. 8th International Conference on Software Engineering*, pages 189–200, 1985. 52, 60, 63, 64

[39] E. Babbie. *Survey Research Methods*. Cengage Learning, second edition, 1990. 53

[40] HERDC. **Higher Education Research Data Collection (HERDC) specifications for the collection of 2011 data, Australis**. Technical report, Higher Education Research Data Collection (HERDC) Specifications, 2014. 53

[41] E. Marcos and A. Marcos. **An Aristotelian Approach to the Methodological Research: a Method for Data Models Construction.** In *Proc. of the 4th. UKAIS Conference. In Information Systems- The Next Generation*, pages 532–543, 1999. 56, 60

[42] W. G. Vincenti. *What Engineers Know and How They Know It. Baltimore*. John Hopkins University Press, new ed. edition, 1990. 56

[43] M. Shaw. **Prospects for an engineering discipline of software**. *IEEE Software*, **7**(6):15–24, 1990. 56, 60

[44] I. Benbasat and R. Zmud. **The Identity Crisis within the IS discipline: Defining and Communicating the Disciplines's core properties**. *MISQ*, **27**(2):183–194, 2003. 56

[45] H. Klein and R. Hirschheim. **Crisis in the IS Field: A Critical Reflection on the State of the Discipline**. *Journal of the Association for Information Systems*, **4**(1):237–293, 2003. 56, 74

[46] B. I. Blum. *Beyond Programming: To a New Era of Design*. Oxford University Press, first edition, 1996. 57

[47] IEEE Computer Society and ACM SE Coordinating Committee. *SWEBOK:Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society Press, 1 edition, 2014. 57

[48] R. L. Glass, I. Vessey, and V. Ramesh. **Research in Software Engineering: an analysis of the literature**. *Information and Software Technology*, **44**(8):491–506, 2002. 58, 60, 74

[49] A. R. Hevner and S. T. March. **The Information System Research Cycle**. *IEEE Computer*, **36**(11):111–113, 2003. 58

[50] D. G. Gregg, U. R. Kulkarni, and A. S. Vinzé. **Understanding the Philosophical Underpinnings of Software Engineering Research in Information Systems**. *Information Systems Frontiers*, **3**(2):169–183, 2001. 58, 60, 74

[51] Jr. Frederick P. Brooks. **Grasping Reality Through Illusion – Interactive Graphics Serving Science.** In *Proc. ACM SIGCHI Human Factors in Computer Systems Conference (CHI '88)*, pages 1–11, 1988. 59, 70

[52] W. Newman. **A preliminary analysis of the products of HCI research, using pro forma abstracts**. In *Proc. ACM SIGCHI Human Factors in Computer Systems Conference (CHI '94)*, pages 278–284, 1994. 59

[53] J. N. Buxton and B. eds. Randell. **Software Engineering Techniques**. *NATO Science Committee, Brussels, Belgium*, 1969. 60

[54] W. F. Tichy, P. Lukowicz, Prechelt L., and Heinz E. **Experimental Evaluation in Computer Science: A Quantitative Study**. *Journal of System Software*, **28**(1):9–19, 1995. 60, 64, 74

[55] F. Tichy Walter. **Should computer scientists experiment more? 16 reasons to avoid experimentation**. *IEEE Computer*, **31**(5):32–40, 1998. 60, 64, 74

[56] M. V. Zelkowitz and D. Wallace. **Experimental models for validating technology**. *IEEE Computer*, **31**(5):23–31, 1999. 60, 64, 74

[57] M. D. Myers. **Qualitative Research in Information Systems**. *MIS Quarterly*, **21**(2):241–242, 2002. 60, 74

[58] P. J. Dobson. **The Philosophy of Critical Realism-An Opportunity for Information Systems Research**. *Information Systems Frontiers*, **3**(2):199–210, 2001. 60, 74

[59] M. V. Zelkowitz and D. Wallace. **Experimental validation in software engineering**. *Information and Software Technology*, **39**(11):735–744, 1997. 60, 64, 74

[60] M. Shaw. **The coming-of-age of software architecture research**. In *Proc. 23rd International Conference on Software Engineering (ICSE 2001)*, pages 656–664, 2001. 60

[61] ICSE-2001 Panel Summary. **Impact Project: Determining the impact of software engineering research upon practice**. In *Proc. 23rd International Conference on Software Engineering (ICSE-2001)*, 2001. 61, 65, 66

[62] A. Mandal. **BRIDGE: A Model for Modern Software Development Process to Cater the Present Software Crisis**. In *Proc. of IEEE International Advance Computing Conference (IACC 2009) Patiala, India*, pages 494–500, 2009. 62, 63, 80, 85, 94, 95, 116, 120, 121, 127, 128, 136, 142, 151, 156

[63] M. Shaw. **Writing Good Software Engineering Research Papers**. In *Proc. of the 25th IEEE International Conference on Software Engineering*, pages 726–736, 2003. 63, 68

[64] S. L. Jackson. *Research Methods and Statistics: A Critical Thinking Approach*. Belmont, CA: Wadsworth, third edition, 2009. 65

[65] N. Juristo and A. M. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic Publisher, first edition, 2001. 66, 73

[66] M. B. Miles and A. M. Huberman. *Quality Data Analysis: A sourcebook of New Methods*. SAGE. NewBury Park-CA, second edition, 1984. 66, 73, 74

[67] T. D. Cook and D. T. Campbell. *Quasi-experimentation: Design and analysis issues for field settings*. IL: Rand-McNally., chicago edition, 1979. 71

[68] J. H. Fetzer. *Philosophy of Science*. Paragon House. United States., first edition, 1993. 73

[69] E. Marcos. **Investigación en Ingeniería del Software vs Desarrollo Software**. In *In the Proc. of Actas de 1er Workshop en Métodos de Investigación y Fundamentos Filosóficos en IS y SI*, pages 136–149, 2002. 73

[70] C. Et al Wohlin. *Experimentation in Software Engineering: An introduction*. Springer, 2012 edition, 2012. 74

[71] Victor R. Basili. **The Experimental Paradigm in Software Engineering**. In *In Experimental Software Engineering Issues: Critical Assessment and Future Directions, International Workshop, Germany, H D Rombach and V R Basili and R W Selby (Eds.), LNCS 706*, pages 3–12. Springer-Verlag, 1992. 74

[72] M. Shaw. **What Makes Good Research in Software Engineering?** *International Journal of Software Tools for Technology Transfer*, **4**(1):1–7, 2002. 74

[73] B. Frakes William and K. Kang. **Software Reuse Research: Status and Future**. *IEEE Transactions on Software Engineering*, **31**(7):529–536, 2005. 76

[74] M. D. McIlroy. **Mass Produced Software Components**. In *Proc. of NATO Software Engineering Conference Report, Garmisch, Germany*, pages 79–85, 1986. 77

[75] A. W. Brown. *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*. Wiley-IEEE Computer Society Press, first edition, 1996. 77

[76] J. Butt. **Reuse rather than rebuilt**. *eWeek*, **18**(44):37–41, 2001. 77

[77] C. Szyperski. *Component Software - Beyond Object-oriented Programming*. Addison Wesley, second edition, 2011. 77, 80

[78] K. K. Lau and Z. Wang. **Software Component Models**. *IEEE Transactions on Software Engineering*, **33**(10):709 – 724, 2007. 78

[79] C. Szyperski. *Component Software*. Addison-Wesley, second edition, 2002. 78

[80] George T. Heineman and William T. Councill. *Component-based Software Engineering - Putting the Pieces Together*. Addison Wesley, first edition, 2001. 78

[81] P. Herzum and O. Sims. *Business Component Factory- A Comprehensive Overview of Component-based Development for Enterprise*. John Wiley, first edition, 2000. 78, 81

[82] A. W. Brown. *Large-Scale, Component-Based Development*. Prentice-Hal, first edition, 2000. 78

[83] Dpunkt Verlag. *WCOP'96 Summary in ECOOP'96 Workshop Reader*. Springer -Verlag, first edition, 1997. 78

[84] A. Deimel, J Henn, and et al. **What characterizes a (software) component?** *Software - Concepts and Tools*, **19**(1):49–56, 1998. 79

[85] M. R. V. Chaudron and E. de Jong. **Components are from Mars.** In *In Proc. 15 IPDPS 2000 Workshops on Parallel and Distributed Processing, Lecture Notes In Computer Science*, pages 727–733, 2000. 79

[86] M. Sparling. **Is there a Component Market**, 2000. 79

[87] G. Gossler and J. Sifakis. **Composition for component-based modeling**. *Science of Computer Programming*, **55**(1-3):161–183, 2005. 80

[88] N. Medvidovic and R. N. Taylor. **A classification and comparison framework for software architecture description languages.** *IEEE Transactions on Software Engineering*, **26**(1):70–90, 2000. 80

[89] R. Roshandel, B. Schmerl, N. Medvidovic, D. Garlan, and D. Zhang. **Understanding tradeoffs among different architectural modeling approaches**. In *Proc. of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA04)*, 2004. 80

[90] J. G. Schneider and O. Nierstrasz. *Software Architectures- Advances and Applications Chapter: Components, Scripts and Glue*. Springer London, first edition, 2000. 80

[91] D. Hybertson. **A uniform component modeling space**. *Informatica*, **25**(0):475–482, 2001. 80

[92] R. W. Jensen. **An Economic Analysis of Software Reuse**. *CROSSTALK The Journal of Defense Software Engineering*, **17**(2):4–8, 2004. 80

[93] W.C. Lim. **Effect of reuse on quality, productivity and economics**. *IEEE Software*, **11**(5):23–30, 1994. 80, 81, 82

[94] P. E. Pinto. **Promoting Software Reuse in a Computer Setting**, 2002. 80, 81

[95] A. Patrizio. **'The New Developer Portals - Buying, selling, and building components on the web speeds companies' time to market**. *Information Week*, page 81, 2000. 80

[96] W. C. Lim. **What is Software Reuse?**, 2002. 81

[97] E. Henry and B. Faller. **Large-scale industrial reuse to reduce cost and cycle time**. *IEEE Software*, **12**(5):47–53, 1995. 81, 82

[98] E. Scannell. **Web Services Reignite Component Reuse**, 2002. 82

[99] K. Kaur, P. Kaur, J. Bedi, and H. Singh. **Towards a Suitable and Systematic Approach for Component Based Software Development**. *International Journal of Computer, Control, Quantum and Information Engineering*, **1**(3):590–593, 2007. 82

[100] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing legacy systems: Software Technologies, Engineering Processes and Business Practices*. Addison-Wesley Professional, first edition, 2003. 82

[101] Sun Microsystems. **Enterprise JavaBeans(TM) specification 2.0. Sun Developer Network Enterprise JavaBeans Technology Official Website. Sun Microsystems, Inc.: Santa Clara CA, 2005; 572 pp.**, 2005. 83

[102] Object Management Group. **CORBA component model (CCM) 3.0. Object Management Group Working Group Specification, Needham MA 2002**, 2002. 83

[103] D. Rogerson. *Inside COM*. Microsoft Press: Redmond WA, pap/cdr edition, 1997. 83

[104] C. Canal and A. Cansado. **Component Reconfiguration in Presence of Mismatch**. *Informatica*, **35**(1):29–37, 2011. 83

[105] A. M. Davis, E. H. Bersoff, and E. R. Comer. **Strategy of Comparing Alternative Software Development Life Cycle Models**. *IEEE Transaction on Software Engineering*, **14**(10):1453–1461, 1988. 85, 145

[106] M. Poppendieck and T. Poppendieck. *Lean Software Develop-ment: An Agile Toolkit*. Addison-Wesley, 2003. 86

[107] Bender Inc. RBT. **Systems Development Life Cycle: Objectives and Require-ments**. Technical report, Bender RBT Inc., 203. 86

[108] M. Cataldo and D. Herbsleb James. **Coordination Break-downs and Their Impact on Development Productivity and Software Failures**. *IEEE Transactions on Software Engineering*, **39**(3):343 – 360, 2013. 88

[109] W. S. Humphrey. **Characterizing the Software Process: A Maturity Framework**. Technical Report SEI-87-TR-11, ESD-TR-87-112, Software Engineering Institute, 1987. 89, 92, 96

[110] M. Walton. **Strategies for Lean Product Development**. Technical Report WP99-01-91, Massachusetts Institute of Technology, 1999. 89

[111] Larman and Vodde. *Scaling Lean and Agile Development: Suc-cessful Large, Multisite & Offshore Products with Large-Scale Scrum*. Addison-Wesley, first edition, 2008. 89

[112] C.C. CHAN Keith and M. L. CHUNG LAWRENCE. **Integrating Process and Project Management for Multi-Site Software De-velopment**. *Annals of Software Engineering*, **14**(1-4):115–143, 2002. 92

[113] M. P. Ginsberg and L. H. Quinn. **Process Tailoring and the software Capability Maturity Model**. Technical Report CMU/SEI-94-TR-024, Software Engineering Institute, 1995. 93

[114] Demming W. Edwards. *Quality, Productivity, and Competi-tive Position*. Massachusetts Institute of Technology, first edition, 1982. 93, 96

[115] A. Mandal and S. C. Pal. **Emergence of Component Based Software Engineering**. *International Journal of Advanced Research in Computer Science and Software Engineering*, **2**(3):311–315, 2012. 94, 122, 138

[116] William W. Lowrance. *Of Acceptable Risk: Science and the Determination of Safety*. William Kaufmann, first edition, 1976. 94

[117] W. S. Humphrey. *Managing the software process*. Addison-Wesley, first edition, 1989. 96

[118] A. Antanovich, A. Sheyko, and B. Katumba. **Bottlenecks in the Development Life Cycle of a Feature: A Case Study Conducted at Ericsson AB**. Technical Report 2010:012, University of Gothenburg, 2010. 96

[119] A. mandal. **SRS BUILDER 1.0: An Upper Type CASE Tool For Requirement Specification**. In *Proc. of 4th National Conference Computing for Nation Development (INDIACom-2010)*, pages 419–423, 2010. 97, 143

[120] X. Ferré, N. Juristo, H. Windl, and L. Con-stantine. **Usability Basics for Software Developers**. *IEEE Software*, **18**(1):22–29, 2001. 97

[121] A. Seffah, R. Djouab, and H. Antunes. **Comparing and Rec-onciling Usability-Centered and Use Case-Driven Require-ments Engineering Processes**. In *Proc. of the User Interface Conference, Second Australasian*, pages 132–139, 2001. 97

[122] C. Canal and A. Cansado. **User Centered Design Process Model- Integration of Usability Engineering and Software Engineering**. In *Proc. of INTERACT-2013*, pages 1–3, 2003. 97

[123] M. Eduard and S. Ahmed. **Adoption of Usability En-gineering Methods: A Measurement-Based Strategy**. In *Proc. of I-USED'09*, 2009. 97

[124] J. Holt. **Current practice in software engineering: a survey**. *Computing and Control Engineering Journal*, **8**(4):167 – 172, 1997. 99

[125] Jalota P. *Integrated Approach to Software Engineering,*. Narosa, third edition, 2006. 100, 112

[126] R. Mall. *Fundamentals of Software Engineering*. PHI, second edition, 2008. 101, 109, 123, 149

[127] G. Goth. **Software-as-a-service: The Spark That Will Change Software Engineering?** *IEEE distributed systems*, **9**(7):1–3, 2008. 103

[128] F. Maranzano Joseph, A. Rozsypal Sandra, H. Zimmerman Gus, W. Warnken Guy, and E. Wirth Patricia. **Architecture Reviews: Practice and Experience**. *IEEE Software*, **22**(2):34 – 43, 2005. 111

[129] S. Schach. *Software Engineering*. Aksen Association, seventh edition, 1990. 112

[130] J. Cho. **A Hybrid Software Development Method For Large-Scale Projects: Rational Unified Process With Scrum**. *Issues In Information Systems*, **10**(2):340–348, 2009. 115

[131] James E. Purcell. **Comparison of Software Development Lifecycle Methodologies**, Accessed in 2015. 115

[132] K. Beck, M. Beedle, and et al. **Manifesto for Agile Software Development**, 2009. 115, 117, 118

[133] R. Vijayasarathy Leo. **Agile Software Development: A Survey of Early Adopters**. *Journal of Information Technology Management*, **19**(2):1–8, 2008. 116

[134] B. W. Boehm and R. Turner. **Observations on balancing discipline and agility**. In *Proc. of the IEEE Agile Development Conference 2003*, pages 32–39, 2003. 116

[135] B.W. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison Wesley, Boston, MA., first edition, 2004. 116

[136] S.I. Hashmi and J. Baik. **Software Quality Assurance in XP and Spiral-A Comparative Study**. In *Proc. of International Conference on Computational Science and its Applications, 2007 (ICCSA-2007)*, pages 367–372, 2007. 116

[137] M. Fritzsche and P. Keil. **Agile Methods and CMMI: Compatibility or Conflict?** *e-Informatica Software Engineering Journal*, **1**(1):9–26, 2007. 116

[138] W.H. Theunissen, D.G. Kourie, and B.W. Watson. **Standards and agile software development**. In *Proc. of the 2003 Annual Research Conference of the South African institute of Computer Scientists and Information Technologists on Enablement Through Technology*, pages 178–182, 2003. 116

[139] R. Turner. **Agile Development: Good Process or Bad Attitude?** *Lecture notes in computer science*, **2559**(0):134–144, 2002. 116

[140] R. Turner and A. Jain. **Agile meets CMMI: Culture clash or common cause?** *Lecture notes in computer science*, **2418**(0):153–165, 2002. 116

[141] B. Boehm and R. Turner. **Using risk to balance agile and plan-driven methods**. *IEEE Computer*, **36**(6):57–66, 2003. 116

[142] J. Nawrocki, L. Olek, M. Jasinski, B. Paliswiat, B. Walter, B. Pietrzak, and P. Godek. **Balancing agility and discipline with xprince.** *Lecture Notes in Computer Science*, **3943**(1):266–277, 2006. 116

[143] J. Nandhakumar and J. Avison. **The Fiction of Methodological Development - a Field Study of Information Systems Development**. *Information Technology and People*, **12**(2):175–191, 1999. 116

[144] D. P. Truex, R. Baskerville, and Travis J. **A methodical Systems Development: the deferred meaning of systems development methods**. *Accounting, Management and Information Technologies*, **10**(1):53–79, 2000. 117

[145] K. E. Wiegers. **Read My Lips: No New Models**. *IEEE Software*, **15**(5):10–13, 1998. 117

[146] R. Baskerville, L. Levine, J. Pries-Heje, B. Ramesh, and S. Slaughter. **Is Internet-speed software development different?** *IEEE Software*, **20**(6):70–77, 2003. 117

[147] K. Conboy. **Agility From First Principles: Reconstructing the Concept ofAgility in Information Systems Development**. *Information Systems Research*, **20**(3):329—354, 2009. 117, 118, 120

[148] VersionOne. **8th Annual State of Agile Survey**, 2013. 118

[149] B. Fitzgerald, K. Stol, R. O'Sullivan, and D. O'Brien. **Scaling Agile Methods to Regulated Environments: An Industry Case Study**. In *Proc. of 35th International Conference on Software Engineering (ICSE), San Francisco*, 2013. 118, 120

[150] O. Cawley, X. Wang, and I. Richardson. **Lean/agile software development methodologies in regulated environments–state of the art**. In *Proc. of International Conference of Lean Enterprise Software and Systems*, 2010. 120

[151] R. France Turk and B. Rumpe. **Assumptions underlying agile software development processes**. *Journal of Database Management*, **16**(4):62–87, 2005. 120

[152] A. Cockburn. *Agile Software Development*. Addison Wesley, Boston, MA, first edition, 2001. 124

[153] S. P. Keider. **Why projects fail**. *Datamation*, **20**(12):53–55, 1974. 128

[154] Y. Saleh and M. Alshawi. **An alternative model for measuring the success of IS projects: the GPIS model**. *Journal of Enterprise Information Management*, **18**(1):47–63, 2005. 128

[155] W. Al-Ahmad, K. Al-Fagih, K. Khanfar, K. Alsamara, S. Abuleil, and H. Abu-Salem. **A Taxonomy of an IT Project Failure: Root Causes**. *International Management Review*, **5**(1):93–106, 2009. 128, 130, 131

[156] The Standish Group International. **The CHAOS Manifesto, 2013: Think Big, Act Small**, 2013. 128

[157] R. Kaur and J. Sengupta. **Software Process Models and Analysis on Failure of Software Development Projects**. *International Journal of Scientific & Engineering Research*, **2**(2):1–4, 2011. 129, 131, 136

[158] Elt. Al Fabriek, Matthias. **Reasons for success and failure in offshore software development projects**, 2014. 130

[159] Alpha Software Inc. **Why Software Projects Fail: A New Assessment of Risk**, Accessed on 5th January 2015. 130, 131, 133, 134

[160] Lorin J. May. **Major Causes of Software Project Failures**, Accessed on 10th Deceber 2014. 131

[161] K. E. Emam, A. Günes, and Koru. **A Replicated Survey of IT Software Project Failures**. *IEEE Software*, **25**(5):84–90, 2008. 131

[162] W. S. Humphrey. **Why Big Software Projects Fail: The 12 Key Questions**. *CROSSTALK: The Journal of Defense Software Engineering*, **18**(3):25–29, 2005. 131

[163] W. S. Humphrey. **Five reasons why software projects fail**, Accessed on: 15/12/14. 131

[164] T. Gilb. **Project Failure: Some Causes and Cures**. *Edited MASTER paper*, **Version: February 29, 2004**(Tom@Gilb.com):1–15, 2004. 131

[165] C. Jones. **Social and Technical Reasons for Software Project Failures**. *CROSSTALK: The Journal of Defense Software Engineering*, **19**(6):2–9, 2006. 131

[166] A. Mandal and S. C. Pal. **Identifying the Reasons for Software Project Failure and Some of their Proposed Remedial through BRIDGE Process Models**. *International Journal of Computer Sciences and Engineering*, **3**(1):118–126, 2015. 136

[167] A. Mandal and S. C. Pal. **Achieving agility through BRIDGE process model: an approach to integrate the agile and disciplined software development**. *Innovations in System and Software Engineering: A NASA Journal*, **11**(1):1–7, 2015. 138, 139

[168] A. Mandal and S. C. Pal. **Investigating and analysing the desired characteristics of software development lifecycle models**. *International Journal of Software Engineering Research and Practices*, **2**(4):9–15, 2012. 139

[169] A. Mandal and S. C. Pal. **A Comparative Analysis of BRIDGE and Some Other Well Known Software Development Life Cycle Models**. *International Journal of Computer Science and Engineering Technology*, **5**(3):196–202, 2014. 139

[170] Chandak S. Sharad and V. Rangarajan. **Flexibility in Software Development Methodologies: Needs and Benefits (Executive Summary)**, June 2013. 142

[171] T. Grance, J. Hash, and M. Stevens. **Security Considerations in the Information System Development Life Cycle**. Technical Report 800-64 REV. 1, NIST SPECIAL PUBLICATION, 2003. 143

[172] L. Alexander and A. Davis. **Criteria for Selecting Software Process Models**. In *Proc. of presented at COMPSAC*, pages 521 –528, 1991. 145

[173] M.N. M. Ali and A. Govardhan. **A Comparison Between Five Models of Software Engineering**. *International Journal of Computer Science Issues*, **7**(5):94–101, 2010. 145

[174] E. Comer. **Alternative Software Life Cycle Models**. In *Proc. of International Conference on Software Engineering*, 1997. 145

[175] P. Dholakia and D. Mankad. **The Comparative Research on Various Software Development Process Model**. *International Journal of Scientific and Research Publications*, **3**(3):1–5, 2013. 145

[176] H. Hijazi, T. Khdour, and A. Alarabeyyat. **A Review of Risk Management in Different Software Development Methodologies**. *International Journal of Computer Applications*, **45**(7):8–12, 2012. 145

[177] S. Malhotra and S. Malhotra. **Analysis and tabular comparison of popular SDLC models**. *International Journal of Advances in Computing and Information Technology*, **1**(3):277–286, 2012. 145

[178] J. Molokken-Ostvold and M. Jorgensen. **A comparison of software project overruns - flexible versus sequential development models**. *IEEE Transactions on Software Engineering*, **31**(9):754–766, 2005. 145

[179] B. A. Sasankar and V. Chavan. **Survey of Software Life Cycle Models by Various Documented Standards**. *International Journal of Computer Science and Technology*, **2**(4):137–144, 2011. 145

[180] S. Taya and S. Gupta. **Comparative Analysis of Software Development Life Cycle Models**. *International Journal of Computer Science and Technology*, **2**(4):536–540, 2011. 145

[181] R. D. Banker and Kauffman R. J. **Reuse and Productivity in Integrated Computer-Aided Software Engineering: An Empirical Study**. *MIS Quarterly*, **15**(3):375–401, 1991. 149

[182] T. Church and P. Matthews. **An Evaluation of Object-Oriented CASE Tools: The Newbridge Experience**. In *Proc. of International Workshop on Computer Aided Software Engineering*, 1995. 149

[183] W. J. Orlikowski. **Division among the Ranks: The Social Implications of CASE Tools for System Developers**. In *Proc. of International Conference on Informationr*, 1989. 149

[184] S. Jarzabek and R. Huang. **The case for User-Centered CASE tools**. *Communications of the ACM*, **41**(8):93–99, 1998. 149

[185] J. Iivari. **Why are CASE Tools Not Used?** *Communications of the ACM*, **39**(10):94–103, 1996. 149

[186] C. F. Kemerer. **How the Learning Curve Affects CASE Tool Adoption**. *IEEE Software*, **9**(3):23–28, 1992. 149

Appendix A: List of Publications

A. Referred Journals

1. **MANDAL A.** and PAL S. C., Achieving agility through BRIDGE process model: An approach to integrate the agile and disciplined software development. *Innovations in System and Software Engineering: A NASA Journal*, 11(1):1–7, 2015.

2. **MANDAL A.** and PAL S. C., Identifying the Reasons for Software Project Failure and Some of their Proposed Remedial through BRIDGE Process Models, *International Journal of Computer Sciences and Engineering (IJCSE)*, 3(1):118–126, 2015.

3. SARKAR S. and **MANDAL A.**, Comparison of Some Classical Edge Detection Techniques with their Suitability Analysis for Medical Images Processing, *International Journal of Computer Sciences and Engineering (IJCSE)*, 3(1):81–87, 2015.

4. SAHA D. , **MANDAL A.** and PAL S. C., User Interface Design Issues for Easy and Efficient Human Computer Interaction: An Explanatory Approach, *International Journal of Computer Sciences and Engineering*, 3(1):127–135, 2015.

5. **MANDAL A.** and PAL S. C., A Comparative Analysis of BRIDGE and Some Other Well Known Software Development Life Cycle Models, *International Journal of Computer Science and Engineering Technology*, 5(3):196–202,2014.

6. DUTTA J., **MANDAL A.** and PAL S. C., A Probabilistic Interval Division Method for Solving Nonlinear Equations, *International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE)*, 3(8):304–310, 2013.

7. GHOSH ROY A., and **MANDAL A.**, RDAT: A tool for DNA Analysis, *Salesian Journal of Humanities and Social Sciences: Technology and Society*, IV(1):56–61, 2013.

8. **MANDAL A.** and PAl S. C., Investigating and analysing the desired characteristics of software development lifecycle models, *International Journal of Software Engineering Research and Practices*, 2(4):9–15, 2012.

9. **MANDAL A.** and PAl S. C., Emergence of Component Based Software Engineering, *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(3):311–315, 2012.

10. **MANDAL A.**, DUTTA J. and PAL S. C., A New Efficient Technique to Construct a Minimum Spanning Tree, *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(10):93–97, 2012.

B. Conference, Seminar and Workshop Papers

11. LAHIRI S. N., DAWN S., MANDAL R. K. and **MANDAL A.**, Steganography based on Image Border Manipulation Technique (IBMT), *Proc. of the National Conference on Computational Technologies-2015 (NCCT'15)*, 231–236, 2015.

12. **MANDAL A.**, GPGPU Computing: Utilizing the GPU of a Computer as General Computational Purpose, *Proc. of Workshop on Application of Information Technology in the Modern Civilization-2013*, St. Joseph's College, Darjeeling, 15th May, 2013.

13. **MANDAL A.**, PAL S. C., Fuzzy Logic and Its Industrial Application, *Proc. of NSAMGT-2012*, Department of Mathematics, University of Kalyani, India, 2012.

14. **MANDAL A.** and PAL S. C., Role of Fuzzy Logic in Computing, *Proc. of UGC-National Seminar on Advances in Mathematics-2012*, Department of Mathematics, University of North Bengal, India, 2012.

15. **MANDAL A.**, SAHA S., SHAW S., UKIL A., An Approach for Image Based Steganography and Cryptography Technique, *Proc. of Research Scholars' Workshop on Computer Science and Application (RSWCSA-2012)*, Jointly organised by CSI, Siliguri Chapter and Salesian College, Siliguri, 2012.

16. **MANDAL A.**, Drivers for Processor Evolution: From Singlecore Towards Multicore, *Proc. of Research Scholars' Workshop on Computer Science and Application (RSWCSA-2012)*, Jointly organised by CSI, Siliguri Chapter and Salesian College, Siliguri, 2012.

17. **MANDAL A.** and MANDAL R. K., Use of e-Resources in Education and Research, *Proc. of UGC (ERO) Sponsored State Level Seminar on Promoting NMEICT-INFLIBNET e-Resources in North Bengal & Sikkim*, 25th Feb, 2011.

18. **MANDAL A.**, SRS BUILDER 1.0: An Upper Type CASE Tool for Requirement Specification, *Proc. of the* 4^{th} *National Conference,INDIACom-2010, Computing for National Development, New Delhi*, February 25–26, 2010.

19. **MANDAL A.** and PAL S. C., An empirical study and analysis of the dynamic load balancing techniques used in parallel computing systems, *Proc. of International conference on Computing and Systems (ICCS-2010) held on Nov.19–20, 2010 at Burdwan University*, 313–318, 2010.

20. **MANDAL A.** BRIDGE: A Model for Modern Software Development Process to Cater the Present Software Crisis, *Proc. of IEEE International Advance Computing Conference (IACC 2009) Patiala, India*, 1(1):494–500, 2009. Also available at IEEE Xplore DOI: 10.1109/IADCC.2009.4809259.

21. SARKAR T., DAS S. C., **MANDAL A.**, A Study of Computer-Based Simulations for Nano-Systems and their types, *Proc. of NATCOM NAMTECH-2009, Lucknow University, India*, 2009.

22. **MANDAL A.**, Operating System as Process Manager, *National Level Technical Paper Presentation Competition-2005*, SVICS-Kadi, 2005.

———— **X** ————

Appendix B: Copies of Some Publications

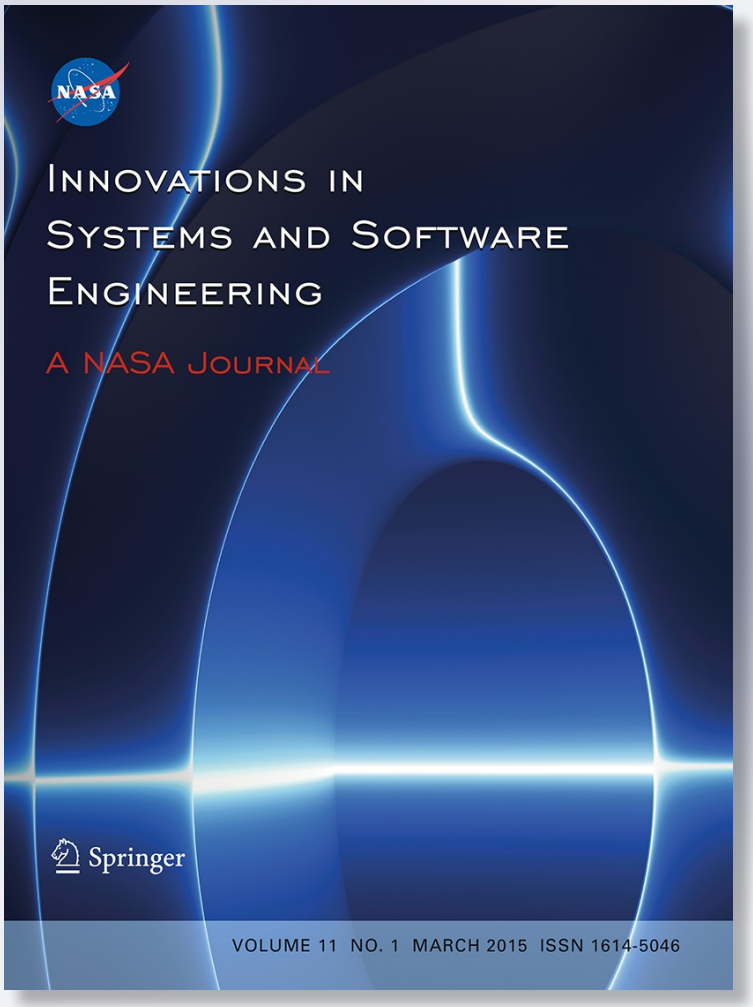
Achieving agility through BRIDGE process model: an approach to integrate the agile and disciplined software development

Ardhendu Mandal & S. C. Pal

Innovations in Systems and Software Engineering
A NASA Journal

ISSN 1614-5046
Volume 11
Number 1

Innovations Syst Softw Eng (2015)
11:1-7
DOI 10.1007/s11334-014-0239-x



Springer

Innovations Syst Softw Eng (2015) 11:1–7
DOI 10.1007/s11334-014-0239-x

ORIGINAL PAPER

Achieving agility through BRIDGE process model: an approach to integrate the agile and disciplined software development

Ardhendu Mandal · S. C. Pal

Received: 14 January 2014 / Accepted: 13 August 2014 / Published online: 23 August 2014

Abstract Despite of many criticisms, agile software development (Beck et al., Manifesto for Agile Software Development, http://agilemanifesto.org, [1]) philosophy has been proved to be quite successful to increase the project success rate up to a significant extent. The primary reasons behind the criticism of agile are the strong violation of the traditional disciplined software engineering theories, principles and practices. The goal of this research paper is to establish the fact that agility may also be achieved by following traditional process models especially with BRIDGE (Mandal, 2009 IEEE International Advance Computing Conference (IACC 2009) Patiala, India, [2]) process model. Hence, the aim of this paper is to integrate the agile and traditional disciplined software development process models and establish the compatibility between both the approaches. At the beginning, we have explored the objectives and principles of agile software development. Next, we briefly discussed the BRIDGE process model and further justified that—following BRIDGE process model, the philosophy of agile may also be achieved and conclude that BRIDGE model has both the capability of traditional software development process model as well as of the agile process.

Keywords Software engineering · Software development lifecycle · Process model · BRIDGE model · Agile software development

A. Mandal (✉) · S. C. Pal
Department of Computer Science and Application, University of North Bengal, Raja Rammohanpur, P.O. North Bengal University, Darjeeling 734013, West Bengal, India
e-mail: am.csa.nbu@gmail.com

S. C. Pal
e-mail: schpal@rediffmail.com

1 Introduction

Many software development life cycle (SDLC) process models and approaches have been introduced till date i.e., Waterfall model, Spiral model, Prototype model, Evolutionary development etc [3]. But, rarely any of these models exactly fits as-it-is for modern software development projects [4]. Hence, often instead of a single process model, most industry follows a sandwich or hybrid model i.e., mix-up of different models on demand basis [5]. Despite of this, very often the customers are unhappy with the end product and many of the projects even had to fail! Agile software development philosophy came out to be a successful approach to increase the project success rate up to significant extent while reducing the development time and cost. Despite of its success, many authors have criticized the agile approach as it often violates the basic theories, principles and practices of traditional software engineering. But the ability to meet client needs and the delivery of quality software products within estimated time is the significant benefits of agile development and is the key to its survival. Some of the additional benefits of agile process are increased productivity, expanded test coverage, improved quality, fewer defects, reduced development time and costs, delivery of better understandable and maintainable code, improved morale, better collaboration, and higher customer satisfaction as pointed out by Vijayasarathy [6]. But as said by Boehm et al. [7], neither the agile nor the traditional disciplined approach alone provide the ultimate approach. Further Hashmi et al. [8] says, there are on-going debates whether the quality of the products of the agile approaches is satisfactory. In addition, Fritzsche et al. [9] and Theunissen et al. [10] mentioned, some projects e.g., safety-critical projects require standards to be followed when developing software. Offhand, the two seem contradicting [11], [12], but several researchers agree that a software project needs

both agility and discipline [7], [13], [14]. New SDLC models are introduced at regular basis as new technology and new research results are needed to be accommodated over the time for modern project needs and suitability. This was the primary philosophy behind the introduction of BRIDGE SDLC model by Mandal [2].

2 Scope of this study

In this paper, we have just considered the theoretical aspects for the purpose of analysis and as evident. We did not consider any real development situations or data for this instance. Presently, we are running three experimental projects in our department in parallel to discover the impact of the BRIDGE process model. But as it shall take more time to complete the projects and to evaluate the successfulness, we simply skipped this for the time being. The real development situations will be disclosed in the future work when the projects will be completed and the experimental results shall be available to us.

3 From traditional disciplined SW development approach towards agile philosophy

3.1 Limitations of the traditional development models

We have many traditional SDLC models i.e., Classical Waterfall Model, Iterative Waterfall model, Spiral model, RAD model etc. in our hand by the time, but a very few are really used in practice exactly as it is. There exist several criticisms about these traditional models. According to Nandhakumar and Avison [15], traditional methods are too mechanistic to be used in detail. Truex et al. [16] pointed out that traditional SDLC models are more dogmatic and claim that traditional methods are merely unattainable ideals and hypothetical "straw men" that provide normative guidance to utopian situations. Further, Wiegers [17] noticed that, industrial software developers have become skeptical about "new" solutions that are difficult to grasp and thus remain not used. Baskerville et al. [18] claim that "to compete in the digital economy, companies must be able to develop high-quality software systems at "Internet speed"—that is, deliver new systems to customers with more value and at a faster pace than ever before". The primarily perceived limitations of the traditional development models are:

- There are too much work associated with documentation.
- They are too sequential.
- They require too much of planning activities.
- It does not show results until the end.
- It engages stakeholders too late.
- Delay in project delivery.
- Increased project cost.

For these reasons, the traditional software development approaches are rarely used in industries these days as they lack suitability for the purpose.

3.2 Agile SW development philosophy: the necessity and its origin

Agile principles evolved to address the above criticisms and primary limitations of the traditional software development. Agile software development is neither a set of tools nor a single methodology. Rather, agile is a philosophy appeared as recommendations in 2001 with an initial 17 signatories. While the publication of the "Manifesto for Agile Software Development" [1] did not start the move to agile methods, which had been going on for some time, it did signal industry acceptance of agile philosophy. Further, Kieran Conboy [19] in his paper has brilliantly derived the functional definition of agile as "the continual readiness of an information system development method to rapidly or inherently create change, proactively or reactively embrace change, and learn from change while contributing to perceived customer value (economy, quality, and simplicity), through its collective components and relationships with its environment".

Agile was a significant departure from the heavyweight document-driven traditional software development methodologies in general used at the time. Agile software development stresses rapid iterations, small and frequent releases, and evolving requirements facilitated by direct user involvement in the development process. In this way, development of agile methods could be seen as cumulative methods built on existing traditional methods where the 'good' parts are kept and the 'bad' parts are omitted or modified.

As per the recent survey report on state of agile by Versionone [20] shows that 88 % of the respondent organizations are practicing agile development, 52 % of the projects are being developed following agile.

From the Agile Manifesto [1], [21] and the definition of agility cultivated by Conboy [19], we may extract and combine the following primary feature of agile methods to be put on focus:

a. Individuals and interactions.
b. Working software delivery.
c. Customer collaboration.
d. Rapid system change incorporation i.e., responding to change.
e. Economic development.
f. Quality product development.

g. Simplicity.
h. Enhance knowledge from change incorporation.

In the next section, we give the list of principles of agile development philosophy with brief outline.

4 Principle of agile development

There were twelve basic principles of agile software development as highlighted in the Agile Manifesto [1]. The detail discussions of these principles are beyond the scope of this paper. But for the purpose of justification and comparison, we combine and highlight these principles from Agile Manifesto [1] and as extracted from Conboy's [19] definition here in brief:

1. Customer satisfaction: The highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Incorporation of rapid system change: Agile methodology welcomes changing requirements even late in development. Agile processes harness change for the customer's competitive advantage.
3. Frequent working SW delivery: Deliver working software frequently—from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Continuous cooperation of client and developer: Business people and developers must work together daily throughout the project.
5. Motivated trusted individuals: Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. Continuous improvement-arrangement of face-to-face conversation: The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Progress measurement: Working software is the primary measure of progress.
8. Sustainable development: Agile processes promote sustainable development. The stakeholders, developers, and users should be able to maintain a constant pace indefinitely.
9. Attention to technical excellence: Continuous attention to technical excellence and good design enhances agility.
10. Simplicity: If the design and implementation are simple, testing is easier and more effective.
11. Self-organizing teams: The best architectures, requirements, and designs emerge from self-organizing teams.
12. Internal assessment for knowledge enhancement: At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
13. Quality assurance: The organization must ensure the quality of the system developed following the standard quality improvement practices.
14. Economic development: Economic development should be achieved through optimum utilization of the resources through lean system development [19].

Some authors [21], [22] have done study about scaling agile methods in regulated environments. But practicing agile in regulated environments will imply many constraints to the agile process that may be either against the philosophy of agile or it could be equivalent to a traditional development approach. As pointed out by Fitzgerald et al. [21], "Some of the essential characteristics of agile approaches appear to be incompatible with the constraints imposed by regulated environments". They further mentioned in the same paper that agile software development methods are faced with some fundamental challenges in regulated environments as a core characteristic of regulated environments is the necessity to comply with formal standards, regulations, directives and guidance. Further, as mentioned by Turk et al. [23] that agile methods and regulated environments are often seen as fundamentally incompatible. Thus, it may lead the agile process to be unlike a traditional process that is not the objective of agile always. The objective should not be to go away from the agile philosophy rather to be adhered to the Agile while achieving the advantages of disciplined approach. It is better not to tailor the agile methods to achieve the discipline necessary in regulated environments, but to optimally achieve the objectives of agile through some disciplined approach in regulated environment.

In the following section, we review the BRIDGE model as for reference and then explain how to achieve agility through this model.

5 Introduction to BRIDGE: a model for modern software development process to cater the present software crisis

The BRIDGE process model was introduced and presented by Ardhendu [2] in IEEE IACC, 2009. Although, detailed discussion of the BRIDGE model is beyond the scope of this paper, but we give the schematic diagram in Fig. 1 of the model for the sophisticated readers just for reference with its primary properties.

5.1 The primary properties of BRIDGE model

The primary properties of BRIDGE model are enlisted below [2]:

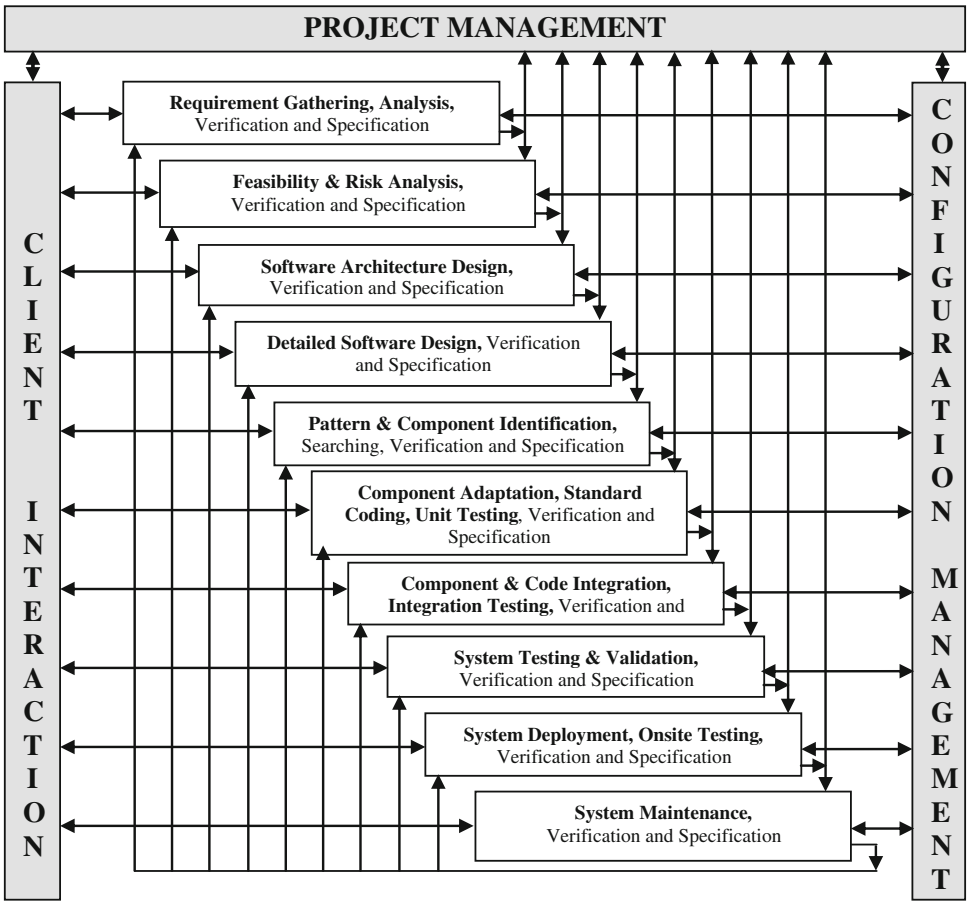


Fig. 1 The BRIDGE Process Model

a. It involves the client over the entire development life cycle activities.
b. It keeps continuous communication among the development team, project management team and client.
c. It enforces explicit verification of individual phases.
d. Supports Components-Based Software Development (CBSD).
e. It enforces on standard coding practices.
f. It considers configuration management as a separate activity.
g. It forces to specify all the phase deliverables.
h. Separate software architecture design phase.
i. Separate system deployment phase.
j. Separate on-site system testing phase.
k. It explicitly instructs to validate the system.

6 Agile development with BRIDGE model: integrating the traditional and agile philosophy

In this section, we are going to explore how the principles of agile can be achieved through BRIDGE process model. We consider all the principles of agile and discuss in brief how these are supported and can be achieved through BRIDGE model.

1. Achieving customer satisfaction: The agile principles proposed customer collaboration for increasing customer satisfaction. In BRIDGE model, this is achieved through continuous client interaction i.e., the left base pillar of the model.
2. Accommodation of requirement change: The initial system requirements may not be mature enough at the very inception of the project. As the developers and client understand the system more and more over the development process, the requirements get refined, even may get modified over the time. As the client is engaged over the entire process, as soon as new requirement is discovered, it is accommodated within the same design or necessary modification is made in the design and its subsequent phases by means of phase iteration. The iterative nature of the BRIDGE process model with the focus on component-based software development facilitates accommodation of requirement changes easily in the system. Promoting system architecture design and component assembly-based system development methodology [24], it is compara-

tively easy following BRIDGE process model to discard, add or modify system requirements.

3. Frequent working SW delivery: As BRIDGE is an iterative process, hence we may start with the initial system requirements and design the initial working software with limited capabilities and deliver to the client at earliest. As the time moves, more and more requirements can be accommodated and delivered to the client in the subsequent software versions and variants.
4. Continuous cooperation of client and developer: This is one of the best features of the BRIDGE model. The client remains as an active member of the development process from the very inception till the end of the entire process. Hence, unlike agile in this model also there exist a close continuous cooperation between the client and developers.
5. Motivated trusted individuals: Alike the client, the management unit also remains as a continuous part over the BRIDGE process model that enables the management to monitor the individual developers. Often, if needed, the management may keep on motivating the developers to give their best and may take necessary actions to make them enough trust worthy for the organization. As a result, over the time, the developers become more motivated trusted individuals for the organization. The management may investigate regularly about the need and working environment that can be allocated optimally so that the job can be done within time and budget while maintaining the quality.
6. Continuous improvement-arrangement of face-to-face conversation: We know face-to-face conversation is the best way to convey and share information. Being an integral part of the process, the management may arrange face-to-face conversation among the developers and even with the client for continuous improvement. As in BRIDGE process model, management team is associated with the development over the development period, the management team may organize such events to promote continuous improvement easily.
7. Progress measurement: In general, before starting any phase it must satisfy some phase entry criterion. As each phase of the BRIDGE model has to go through a strict verification activity before starting the immediate next phase, hence all phases have to satisfy the phase exit criteria too. The phase entry and phase exit criterion is nothing but some intermediate milestones in addition to some other desirable constraints. These milestones may be even partial working software. Through these intermediate milestones, both the development and management team may assess and measure the progress periodically. Following the way, in BRIDGE model, we may measure the progress of the project at regular interval.
8. Sustainable development: Continually evolving, growing, and changing are a natural phenomenon of any organism—none of the other organisms can survive without it. The same thing can be applied to software also, just for analogy. Very few softwares are written once, installed, and then never changed over the course of its lifetime. As per Lehman's first law [25] regarding software, a software product must change continually or become progressively less useful. New requirement will get discovered over time, some old requirements may need to be modified or discarded for the products' survival and its enhancements. Hence, the system has to be designed and developed keeping the view of anticipating the future changes in mind. Following such type of development is what called sustainable development. But unfortunately, it is a rare practice as it may increase the product cost and other development burdens. Maintaining and enhancing software to cope with newly discovered problems or new requirements can take more time than the initial development of the software. Sustainable development requires a singular focus on a form of technical excellence that regularly provides useful software to customers while keeping the cost of change minimal. Under the umbrella of effective management, the project stakeholders can maintain consistent work pace and speed promoting sustainable development following the BRIDGE model as all of the stake holders work together in this model.
9. Attention to technical excellence: In addition to design phase, the BRIDGE model has a specific software architectural design phase. Further, being the customer, an integral part of the development process, technical excellence can be monitored by both the development and project management team. Continuous attention to architectural design, low-level design and technical excellence of BRIDGE model enhances the agility.
10. Simplicity: The notion of simple is very subjective and giving practical guidelines what is simple and how to accomplish that is impossible. At the beginning, requirements seem to be quite complex, but after the analysis and as the project development progresses, they become clear. The developers should only implement features that have been agreed upon with the customers, nothing more. The art of maximizing the amount of work not done is essential. Cockburn and Glass [26] warn that simplicity should not mean neglecting design by starting programming as soon as possible. This principle most strongly supports the design of high-quality architecture and implementation. This principle further acknowledges that in programming, it is more difficult to make simple design than cumbersome solutions. Following a disciplined and systematic approach makes any process simple. The system requirements should be simple and must specify the scope of the project very specific and clear. If the design and

implementation are simple, testing becomes easier and effective. As in BRIDGE process model, all the phases are distinctly well defined, it promotes simple design and implementation of the system that makes the other subsequent activities easier, simple and more effective.

11. Self-organizing teams: By self-organizing team, we mean that the team members share a common goal and belief that their work is interdependent and collaboration is the best way to accomplish their goal. Rather than having a manager with responsibility for planning, managing and controlling the work, the team members share increasing responsibility for managing their own work and also share responsibility for problem-solving and continuous improvement of their work processes. Hence, the empowered team members' reduce their dependency on top management as they accept accountability. The team structure places ownership and controls close to the core of the work. The primary role of the project management is to build individual stakeholder a dedicated responsibility center which is easy to establish through BRIDGE process model. By developing individual responsibility centers, the team may become the self-organizing team.

Advantages of self-organizing:

a. People in a self-organized team are able to make decisions themselves and accordingly adapt to changing situations.
b. Self-organized teams do a much better job of utilizing the talents of the team because more minds are involved in any activity.
c. Self-organized teams have much more communication between team members.
d. The best way to learn is to have actual responsibility and opportunities to do new things.
e. A self-organized team is collectively aware of the upcoming work and much better able to bootstrap themselves with new work when they complete their existing task.
f. Self-organized teams spread knowledge around much better and make decisions together. That makes each team member more effective because they have much more background on the "why" of the coding assignments.
g. A command and control team member often lacks an understanding of why a decision was made because they were not involved with that decision. This may hamper their ability to follow a design or approach which restricts productivity.

As in BRIDGE model, the project management has consistent involvement with the development team, so if needed then the top management can facilitate the development team at any moment to be self-organized.

12. Internal assessment for knowledge enhancement: In association to project management, the individual stakeholders may carryout internal assessment at regular intervals. Through the internal assessment result, the progress may be measured too. Further from the internal assessment, the team may identify the various bottlenecks and upon rectifying and adjusting become more effective and plan the future activities efficiently. Being project management team with the development team in the BRIDGE process model, internal assessment becomes much easier.
13. Quality assurance: In BRIDGE model, quality ensures through implementation of multi level quality improvement methods through phase-wise verification, unit level, integration, and system testing, and system validation. Further, during maintenance, discovered errors if any may be rectified. In addition, being management team working together with the development team, the entire process and individuals remain under proper control and monitoring of the management teams. Overall, the integrated project development environment of this process model ensures the system quality to be achieved and improved.
14. Economic development: Economic development is achieved through optimal utilization of the resources and restricting misuse of resources. The optimal resource management is done through management authority with individual's care and concern. In BRIDGE, all the stakeholders work together with better coordination and concern ensuring economic system development.

7 Conclusion

Despite of the criticism towards traditional software development process, the goodness of agile process is questionable. Agile may not be the good choice for some projects always. Often, traditional process model proves to be better than agile for some types of projects. Hence, our objective should not be to criticize the traditional process models over agile, rather we must carry out research to accommodate the good attributes of agile process in traditional process models. In this paper, we have shown that the philosophy of agile may also be achieved through the BRIDGE process model which follows the principle of traditional software development too. Hence, we recommend BRIDGE process model to be practiced by industries for modern software development projects.

Acknowledgments We would like to thank the advisor of for his/her valued comments that helped us to improve the paper significantly.

References

1. Beck K, Beedle M et al (2001) Manifesto for agile software development. http://agilemanifesto.org
2. Mandal A (2009) BRIDGE: a model for modern software development process to cater the present software crisis. In: IEEE International Advance Computing Conference (IACC 2009) Patiala. IEEE Xplore, India. doi:10.1109/IADCC.2009.4809259
3. Pressman RS (2005) Software engineering: a practitioner's approach, 6th edn. Mc-Grawhil, New york
4. Cho Juyun (2009) A hybrid software development method for large-scale projects: rational unified process with scrum. J Issues Inform Syst X(2):340–348
5. Purcell JE (2007) Comparison of software development lifecycle methodologies. SANS Software Security, San Antonio
6. Vijayasarathy LEOR (2008) Agile software development: a survey of early adopters. J Inform Technol Manage XIX(2):1–8
7. Boehm B, Turner R (2003) Observations on balancing discipline and agility. In: Proceedings of the Agile Development Conference, IEEE Computer Society, Salt Lake City, pp 32–39
8. Hashmi SI, Baik J (2007) Software Quality Assurance in XP and Spiral-A Comparative Study. In: International Conference on Computational Science and its Applications, Proceedings of ICCSA-2007, IEEE, Fukuoka, pp 367
9. Fritzsche M, Keil P (2007) Agile methods and CMMI: compatibility or conflict? e-Informatica. Softw Eng J 1(1):9–26
10. Theunissen WH, Kourie DG, Watson BW (2003) Standards and agile software development. In: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology, South African Institute for Computer Scientists and Information Technologists, Johannesburg, pp 178
11. Turner R (2002) Agile development: good process or bad attitude? Lecture notes in computer science, pp 134–144
12. Turner R, Jain A (2002) Agile meets CMMI: culture clash or common cause? Lecture notes in computer science, pp. 153–165
13. Boehm B, Turner R (2003) Using risk to balance agile and plan-driven methods. IEEE Comput 36(6):57–66
14. Nawrocki J, Olek L, Jasinski M, Paliswiat B, Walter B, Pietrzak B, Godek P (2006) Balancing agility and discipline with xprince. Lecture Notes Comput Sci 3943:266
15. Nandhakumar J, Avison DE (1999) The fiction of methodological development—a field study of information systems development. Inform Technol People 12(2):175–191
16. Truex DP, Baskerville R, Travis J (2000) Amethodical systems development: the deferred meaning of systems development methods. Account Manag Inform Technol 10(1):53–79
17. Wiegers KE (1998) Read my lips: no new models. IEEE Softw 15(5):0–13
18. Baskerville R, Levine L, Pries-Heje J, Ramesh B, Slaughter S (2003) Is Internet-speed software development different? IEEE Softw 20(6):70–77
19. Conboy K (2009) Agility from first principles: reconstructing the concept of agility in information systems development. Inform Syst Res 20(3):329–354
20. VersionOne (2013) 8th Annual State of Agile Survey
21. Fitzgerald B, Stol K, O'Sullivan R, O'Brien D (2013) Scaling agile methods to regulated environments: an industry case study. In: Proceedings of 35th International Conference on Software Engineering (ICSE), San Francisco
22. Cawley O, Wang X, Richardson I (2010) "Lean/agile software development methodologies in regulated environments-state of the art". In: International Conference Lean Enterprise Software and Systems, LNBIP 65
23. Turk R France, Rumpe B (2005) Assumptions underlying agile software development processes. J Datab Manag 16:2005
24. Mandal A, Pal SC (2012) Emergence of component based software engineering. Int J Adv Res Comput Sci Softw Eng 2(3)
25. Mall R (2009) Fundamentals of software engineering 2nd edn, PHI Publication
26. Cockburn A (2001) Agile software development. Addison-Wesley, Boston

International Journal of Computer Sciences and Engineering Open Access
Research Paper Volume-3, Special Issue-1 E-ISSN: 2347-2693

Identifying the Reasons for Software Project Failure and Some of their Proposed Remedial through BRIDGE Process Models

Ardhendu Mandal*
Dept. of Computer Science and Application,
University of North Bengal,
Dist-Darjeeling, India
am.csa.nbu@gmail.com

S. C. Pal
Dept. of Computer Science and Application,
University of North Bengal,
Dist-Darjeeling, India
schpal@rediffmail.com

***Abstract*— There are enough evidences of software project failures. Starting from economic losses to live losses is caused by many software project failures. Software project failures have significant impact on both social and economic factors. Hence, it is important to identify the different reasons for project failures. If these reasons are pre-known, actions can be taken during project development to reduce project failure risks. In this paper we identify and categorized the project failure root causes based on their different sources. Then briefly we have highlighted the primary features of the BRIDGE [1] process model and explored the ways and means how these project failure reasons may be reduced or alleviated by following the BRIDGE process model.**

Keywords- Software Engineering, Project Failure, BRIDGE Process Model, SDLC Model

I. INTRODUCTION

Software project failures are one of the primary reasons for increased cost of software product and services. There are enough evidences of project failures in past and present. Any organizations have to compensate the cost of the failure projects from the success projects. For these reason, software are still beyond the scope of small and medium scale companies causing significant impact on both social and economical factors. Apart from this, starting from economic losses to live losses is also caused by software project failures. Hence, it is important to identify the different reasons for software project failures. If these reasons are pre-known, actions can be taken during project development to reduce project failure risks.

At the beginning we have discussed about the criterion to evaluate a software project to be called successful or failed. Then, we have identified, categorized and briefly discussed different the root causes of project failures based on their source areas. Next, we have briefly highlighted the primary features of the BRIDGE [1] process model and explored the various ways and means to reduced or alleviate these project failure reasons by following the BRIDGE process model.

II. RESEARCH GOAL AND OBJECTIVES

The goal of this work is to identify the different reasons for software project failure and categorization of those reasons based on their originating sources. Further, we have tried to find out the project failure risks especially

originating from software process model and to propose their remedial strategy specially through following the BRIDGE [1] process model.

III. DEFINITION OF SUCCESSFUL AND FAILED SOFTWARE PROJECTS

The primary objective of software engineering is to develop software that agreed upon functionality and:

a. Within Time
b. Within Budget, and
c. With Good Quality

Any software development project that satisfies the above criteria is to be called successful. According to Keider [2] and Saleh [3], a project should deliver agreed upon functionality on time and within estimated budget. Successful software project maybe defined as any software project that is set to support initially-approved functionality, as well as the project comfortably satisfying the stakeholders and being accepted and largely used by the end users after deployment. Hence, Software project failure is defined as any project that is set to support the operations of an organization by exploiting the resources of information technology that fails to deliver the intended output within the originally allocated cost, time schedule [4].

IV. PROJECT FAILURE STATISTICS

To highlight the importance of this study, in this section some statistical data about the software project failure are shared. The survey statistics about software project failure and project estimate overrun carried out by Standish Group International i.e. the CHAOS Manifesto [5], in 2013 are given in Table 1 and Table 2:

Table 1: Project Performance Statistics

Year	Successful	Challenged	Failed
1994	16%	53%	31%
1996	27%	33%	40%
1998	26%	46%	28%
2000	28%	49%	23%
2002	34%	51%	15%
2004	29%	53%	18%
2006	35%	46%	19%
2008	32%	44%	24%
2010	37%	42%	21%
2012	39%	43%	18%

Table 2: Project Estimates Overrun Statistics

Year	Time Overrun	Cost Overrun	% of Features Delivered
2004	84%	56%	64%
2006	72%	47%	68%
2008	79%	54%	67%
2010	71%	46%	74%
2012	74%	59%	69%

From the statistical data presented in Table 1, it is observed that the alterative year average of project successful rate is 30.3%, project challenged by 46% and project failed by 23.4%.

From the statistical data presented above in Table 2, it is observed that the alterative year average of project time overrun rate is 76%, project cost overrun 52.5% and project feature delivery rate is 68.4%.

Rupinder Kaur and Dr. Jyotsna Sengupta in their paper [6] presented the following statistical data:

- As per the Research Report of ESSU (European Service Strategy Unit), 57% of contracts experienced cost overruns, 33% of contracts suffered major delays, 30% of contracts were terminated, and 12.5% of Strategic Service Delivery Partnerships have failed.
- As per the KPMG Survey, on average, about 70 % of all IT-related projects fail to meet their objectives.
- From the presentation on software failure by Bob Lawhorn following statistics are presented:
  - Poorly defined applications (miscommunication between business and IT) contribute to a 66% project failure rate, costing U.S. businesses at least $30 billion every year.

- 60% – 80% of project failures can be attributed directly to poor requirements gathering, analysis, and management.
- 50% are rolled back out of production
- 40% of problems are found by end users
- 25% – 40% of all spending on projects is wasted as a result of re-work.
- Up to 80% of budgets are consumed fixing self-inflicted problems (Dynamic Markets Limited 2007 Study)

Research indicates that more than 50% of all IT projects become runaways--overshooting their budgets and timetables while failing to deliver the expected outcomes [4, 7].

Johnson [4] reported that the overall project success had increased from 16% in 1994 to 28% in 2000.

That makes it very curious, but probably not surprising, that according to an article in the IEEE Spectrum, about 10% of projects are abandoned either before or after completion, because the end product will not actually resolve the original business challenge [8].

V. IDENTIFYING THE COMMON REASONS FOR SOFTWARE PROJECT FAILURE AND THEIR CATEGORIZATION

Often it is easy to identify whether a software project is successful or failed. But, it is really a tough job to identify and understand the actual reasons for project failure. For example, if the delivered system fails to meet the needs of the customer or user, the first question to ask is, "Why?":

- Was it because the development group didn't do a good job? Or
- Perhaps the requirements were not properly gathered or used? Or
- May be the people responsible for supplying the requirements were inaccurate? Or
- Was it something else?

Further, being software development a people intensive job, it is more complex to identify the exact reason to failure and to provide solutions to project failure. Usually, often there are multiple factors causing a software project to fail.

Possible areas/sources of Project Failure Reasons:

Form the above discussions it is easy to understand that there are several possible areas or sources of reasons to project failure. Some of the investigated sources of causes to software project failure are explored and listed below:

- People Sources
- Technology Sources
- Process Sources
- Organizational Sources
- Management Sources
- Business Sources
- Project Sources

Some reasons for project failure are easy to classify as belonging to one area or another, but some are harder to categorize even. So far significant effort has been made to identify and analyze the causes of software project failure discussed below [4, 6, 8, 9, 10, 11, 12, 13, and 14]**.** Now we try to identify the possible project failure reasons from the different sources as identified above.

A. Project Failure Reasons Originating from People Sources

In a software development project typically three types of stakeholders are associated:

- *Users:* Some of the project failure causes originating by these types of people may be due to:
  - Poor User Input
  - Lack of User Training
- *Client:* Some of the project failure causes originating by these types of people may be due to:
  - Conflicts

  - Politics
- *Project Development and Management Team:* Some of the project failure causes originating by these types of people may be due to:
  - Poor-Quality Work by developers
  - Poor-Quality Work by Management Personals

The project failure reasons may originate from one or more of these people sources. *Firstly,* often, the users are either unable to deliver the exact requirements to be delivered by the system or even may not be clear during initial stages of the development. As a result the project may fail due to wrong or inadequate user input. *Secondly,* often the project client and system users are different. Because of poor or misunderstanding, lack of communication gap between them, the client may convey wrong information and requirement to project development team that may lead to project failure. *Thirdly,* the project may fail because of the development team itself. These days, software development teams have become distributed in nature. The lack of communication among the development team and inefficient human resources may become the bottle neck for the project success. *Finally,* insufficient and inefficient project management team may lack to provide necessary management support to the project causing project to fail.

B. Project Failure Reasons Originating from Technology Sources

The rapid technological advancements are often good, but not always. Being software development a time intensive job, very often the technology used for the project implementation becomes obsolete before completion of the project causing the project to fail. Generally, the projects get cancelled before their completion. Further, the technology used if not chosen wrong, may be new and immature failing to perform as expected causing project failure. From technology sources SW project failure may arise due to the following reasons:

- Wrong Technology selection.
- Technology too new or didn't work as expected
- Use of immature technology
- Technology planning

C. Project Failure Reasons Originating from Process Source

Process failure is the largest and potentially the most pernicious of all sources of project failure and has been at the root of problems for decades. If the goal of a process is to produce a specific outcome, then anything that either delays or prevents the achievement of that specific outcome is a form of process failure. The process might deliver something, but if it does not deliver anticipated outcomes or does not meet expectations; the result is a failed process. This form of failure usually leads to finger pointing between development groups and users, with each claiming the other did not understand [8]. The root causes of SW project failure originating from process sources may include the following:

- **Wrong Process Selection:** There are many process models, but all have their own features and limitations. Often not all process models are suitable for any kind of projects. Thus process selection is typically challenging for any project implementation. Wrong or inappropriate process selection may lead to project fail.
- **Lack of User Involvement:** Non involvement of user and customers in the development process is one of the principle reasons that software does not fully meet customer expectations.
- **Lack of Communications:** When we think about communications failure the first thing that comes to mind is, "It's their problem," and it is usually an internal dialog. However, lack of communications with end-user or customers is rarely immediately considered, and it turns out to be one of the major problems. Further, delayed communications or communications latency is blamed as the reason for failure: "They didn't get back to me in time."
- **Unnecessary Processes:** Unnecessary processes apply to wasted or duplicated effort as well as a management or reporting structure that adds "heavy-weight" reporting and accountability to the development process.

- **Careless, sloppy, or missing software development processes:** Sloppy development process is the core value for the software engineering movement, contributed to the acceptance of object-oriented programming, helped fuel the agile movement, and more. Consider the customer at every step in the development process. While that will not guarantee the development process will be free from sloppiness, it will help focus on what is important.
- **Non-adaptability of process to Changes:** More importantly, the presence of one or more of these process failures contribute to business failure if the organization is not able to respond to changing business or market conditions. They also make it difficult to respond to customer-perceived incidents that disrupt service delivery.

D. Project Failure Reasons Originating from Organizational Sources

- New to business- lack or no prior experience
- Improper Organizational Structures in respect to project need
- Poor communication among customers, developers, and users
- Reasons Related to Human Resource
- Insufficient Resources
- Organizational culture and structure

E. Project Failure Reasons Originating from Management Sources

Additional project failure reasons may originate from project management sources. Some of the identified reasons contributing to project failure in this respect are as follows:

- Poor communication among customers, developers and users with management
- Lack of leadership and effective Management
- Poor reporting of the project's status
- Insufficient involvement of Senior management
- Insufficient staff/team Size
- Inaccurate estimates of needed resources
- Lack of proper project management and control
- Sloppy development practices
- Failure to plan
- Commitment and patterns of belief
- Poor quality management and control

F. Project Failure Reasons Originating from Business Sources

In this section we focus on different causes of failures at the business level [8] that directly affect a software development project:

- **Non adaptive to changing conditions:** One of the most obvious forms of business failure also turns out to be the primary reason that *development organizations cannot readily adapt to changing conditions*: specifically, lack of management commitment.
- **Poor selection and use of a particular tool or vendor:** Another potential source of business failure is the management requirement that dictates the use of a particular *tool* or *vendor* without considering the outcomes expected by customers.
- **Commercial pressures:** Often there is commercial pressure on the project from business sources. Time-to-market, competition in business, economic breakdown, economic competency among similar products are the different sources of commercial pressures.

G. Project Failure Reasons Originating from Project Sources

Different projects are of varying nature, types and complexity. Often, there are many intrinsic reasons to the project itself causing the project to fail! These reasons may be related to the system requirements, risks, budget, schedule etc. Some of the project related reasons originating from the project itself are identified and listed below:

- Reasons Originating from System Requirements
  i. Lack of proper understanding and poor definition of system requirements
  ii. Changing system requirements and project scope
  iii. No more need for the system to be developed
- Reasons Related to Project Risk
  i. Poor project risk identification, management and control
  ii. Late project failure warning signals
  iii. Unrealistic or unarticulated project objectives and goals
- Reasons Related to System
  i. Project's complexity
  ii. Poor system architecture and specification
  iii. Critical quality problems with software
- Reasons Related to Budget and Schedule
  i. Inaccurate/over budgeting
  ii. Hidden costs of going "Lean and Mean"
  iii. Unrealistic and over schedule estimation

The Avanade Research Report (2007) [6] disclosed that 66% of failure due to system specification, 51% due requirement understanding, and 49% due to technology selection.

Further, TCS (Tata Consultancy Services) 2007 [6] reported that 62% of organizations experienced IT projects that failed to meet schedules, 49% suffered from budget overruns, 47% had higher-than-expected maintenance costs, 41% failed to deliver the expected business value and ROI, 33% file to perform against expectations.

VI. BRIDGE PROCESS MODELS: A BRIEF HIGHLIGHTS

Although the details discussion of the BRIDGE [1] model is beyond the scope of this paper, just the schematic diagram of the BRIDGE process model is given below in *Figure 1* with its analytical results.

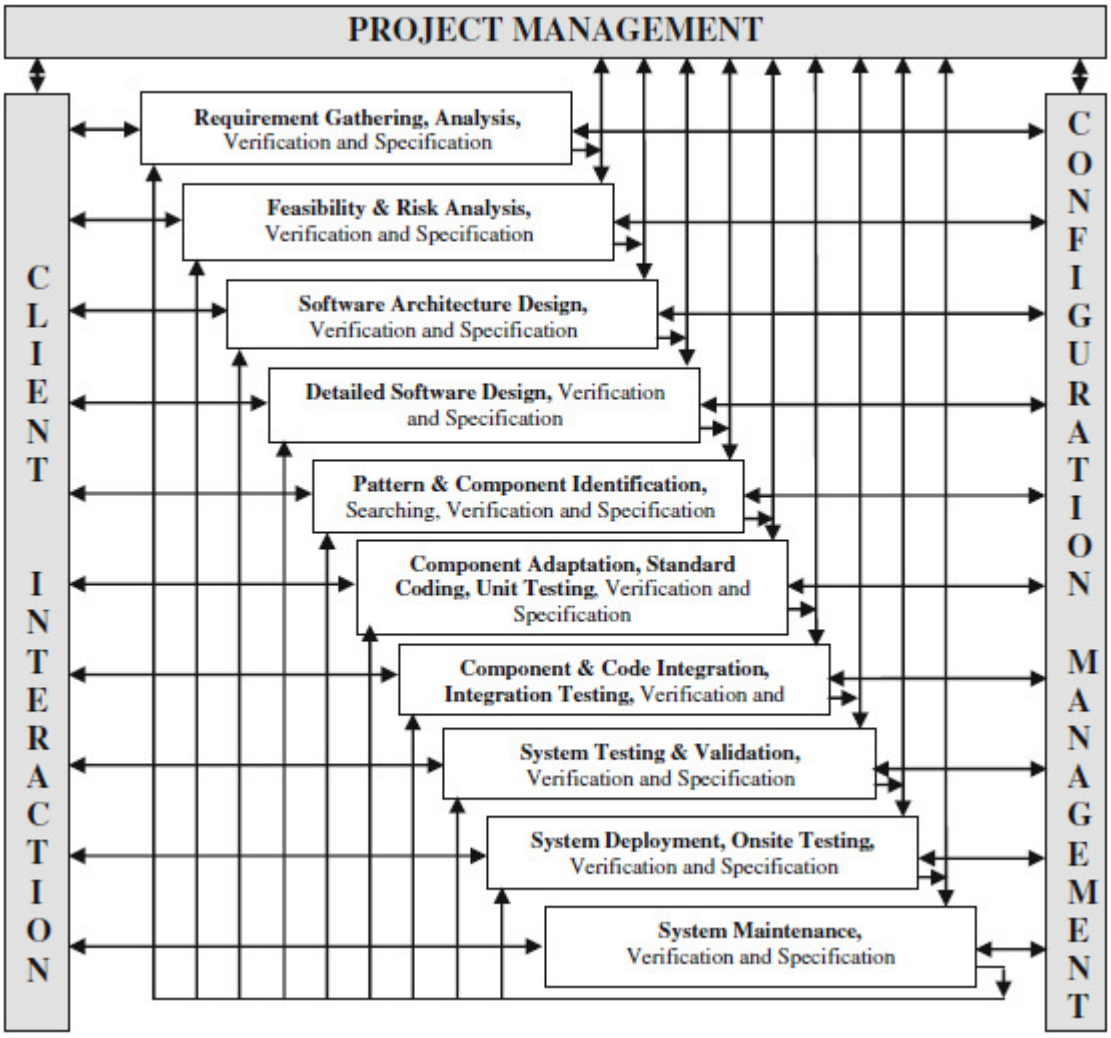


Figure 1: BRIDGE [1] Process Model

The in-depth study of the BRIDGE model discloses a lot of information that may be used to analyze the model. These are briefly discussed below [1, 15, and 16]:

- It involves the client over the entire development life cycle activities.
- It keeps continuous communication with the project management team.
- It explicit verification of individual phases.
- Separate software architecture design phase.
- Separate system deployment phase.
- Separate on-site system testing phase.
- Supports components based software development.
- It emphasizes on standard coding.
- It considers configuration management as a separate activity.
- It forces to specify all the phase deliverables.

- It explicitly instructs to validate the system.

VII. REMEDIAL TO PROJECT FAILURE RISKS TROUGH BRIDGE PROCESS MODEL

As we have discussed in the earlier sections, the project failure reasons may originate from different sources of the project i.e. people, technology, process, organizational, management business and project sources. It is not possible to address all these project failure reasons only by following any process model. However, many of these failure reasons directly or indirectly related to the software development process model. So by following any suitable process model, many of this project risk can be reduced.

In this section we discussed the remedial to some of the project failure reasons identified in the earlier sections by following BRIDGE [1] process model.

A. Remedy to Project Failure Reasons Originating from People Sources

- **Poor User Input:** In most of the process models, users are involved only during the initial phases of the development process when often the system requirements are unclear and ambiguous. Hence, the user inputs are often incorrect and poor. As the development proceeds, the requirements start getting clear. But by these times, the users are not a part of the development process. Hence, user input remains poor causing risk to project success. As in BRIDGE, the user are involved over the entire development process, the user remains the scope to provide update inputs that increases the rate of project success.
- **Lack of User Training:** Many of the organizations do think that only development and delivery of the system is the only responsibility of them. Thus they don't often take user training as serious part or their interest. But the truth is that if the users are unable to use the system easily and efficiently, the project fails irrespective of how good the developed system may be! Proper and good documentations are the key to user training but are often ignored by many organizations. In BRIDGE, special focus is given on documentations at different phases of the development process. Thus simultaneously at the end of all phases proper documents are produced that may help during the user training process and self learning.
- **Client Conflicts and Politics:** Both of these issues arise because of the lack of user involvement. The scope of these problems may be reduces only by involving the users in the development process making the user themselves to be an individual responsibility centers in the project, which is supported in BRIDGE process model.
- **Poor-Quality Work by developers and Management Personals:** There are two possible reasons for this problem to arise:
  - *Lack of Knowledge, Skill and Expertise of Developers:* In this case, the process model doesn't have any role to play; rather it is an organizational staffing and human resource related problem to be managed at organizational level.
  - *Because of Lack of Tendency to Quality-Work of Developers:* However, this reason may be handled by proper monitoring and management control efficiently by following BRIDGE as project management team is always with the development team.

B. Remedy to Project Failure Reasons Originating from Process Sources

- **Remedial to Wrong Process Selection:** The basic reasons for selecting wrong process model are unclear process objectives and goals. Further, as the different features of any process models are not distinct and ambiguous, people often selects wrong process model. In BRIDGE, the feature of the process model is very clear and unambiguous; the concerned may judge the suitability of this model for any typical project easily.
- **Remedy to Lack of User Involvement:** One of the primary features of the BRIDGE process model is the involvement of client over the entire development process that alleviates the project failure risks.
- **Lack of Communications:** This problem may also be originated from management sources. Often in no process model except BRIDGE all the stakeholders' including project management team works together. Working together by different project stakeholder in BRIDGE, the communication gap is reduced among them.

C. Remedy to Project Failure Reasons Originating from Management Sources

Remedial support to project failure causes originated from management sources are beyond the scope of any process model. For example, the quality and expertise level of the individual management and development personals don't comes under the scope of development process. Thus risks i.e. lack of leadership and effective management, poor reporting of the project's status, insufficient involvement of senior management, insufficient staff/team size, inaccurate estimates of needed resources, lack of proper project management and control, sloppy development practices, failure to plan, commitment and patterns of belief, poor quality management and control etc. depends on the quality and effective management team.

However, given an effective project management team, due to lack of direct involvement to the development process, some of the above problems may also arise, but in BRIDGE working all together under same umbrella automatically gets reduced.

D. Remedy to Project Failure Reasons Originating from Project Sources

- **Alleviating Reasons Originating from System Requirements Sources:** In BRIDGE, to alleviate reasons relate to lack of proper understanding and poor definition of system requirements, there is a dedicated phase for requirement gathering, analysis and specification that has to satisfy both the phase entry and phase exit criteria to get qualified. Further, being customer in BRIDGE always available to system analyst and developers there remains a scope to clear the doubts related to system requirements over the development process. It is also known that we may achieve the agile philosophy following BRIDGE process model [16]. Thus accommodation and adaption of changing requirement becomes easy following this process model. Moreover, as BRIDGE process model promotes Component Based Software Development (CBSD) approach [17], changing project scope becomes easy by unplugging and plugging additional software components providing services as demanded. But in case of no more need for the system to be developed, no process model can help at all, as the case with BRIDGE.
- **Alleviating Reasons Related to Project Risk:** To alleviate risks related to risk identification, management and control, and late project failure warnings signals, in BRIDGE one phase is dedicated to feasibility study and risk analysis. Further, verification activity at the end of the individual phases helps to identify and reduce these types of project failure risks.
- **Reasons Related to System Complexity:** The well known tool and technique to manage project complexity is abstraction. Using software components and CBSD approach [16], BRIDGE has the quality to handle project complexity issues. In relation to poor system architecture and specification issues, to promote CBSD, in BRIDGE there is a distinct phase for architectural design of the system apart from detailed design and at the end of the all individual phases forceful specification is mandatory.
- **Related to Software Quality Assurance:** To ensure quality system development irrespective of human related issues, BRIDGE recommends to perform verification at the end of each development phases and to perform validation and testing of the system before system deployment. Further, to ensure quality development, the organizations additionally may follow the guidelines and recommendation given by different standard bodies i.e. SEI, ISO, and Six-Sigma etc. to attain different CMM levels, ISO certifications etc.

The remedy to other project failure reasons/risks originating from other difference sources i.e. technology, organization, business are beyond the scope of capability of any process model. Further, problem related to project budget and scheduling depends heavily on the degree of expertise level of the project manager/estimator and are beyond the capability scope of any process model as the case with BRIDGE.

VIII. CONCLUSION

In this paper we have identified the different reasons contributing to project failure from there originating sources. Then we have highlighted the features of BRIDGE process model and discussed at length on how some of these project failure reasons may be reduces following BRIDGE process model. The comparative analysis of BRIDGE with some other well known process model explored the distinguished features of BRIDGE over other

[15, 18]. Thus, we conclude by recommending the BRIDGE process model to be followed for SW development projects to gain project success rate.

REFERENCES

[1] Mandal A., BRIDGE: A Model for Modern Software Development Process to Cater the Present Software Crisis, Proc. IEEE Int'l Conf. Advance Computing Conference, 2009. [Also available at IEEEXplore, DOI: 10.1109/ IADCC.2009.4809259], 494-500, 2009.

[2] Keider, S.P., Why projects fail. Datamation, 53-55, 20(12), 1974.

[3] Saleh, Y., & Alshawi, M., An alternative model for measuring the success of IS projects: the GPIS model, Journal of Enterprise Information Management, 47-63, 18(1), 2005.

[4] Walid Al-Ahmad, Khalid Al-Fagih, Khalid Khanfar, Khalid Alsamara, Saleem Abuleil, Hani Abu-Salem, A Taxonomy of an IT Project Failure: Root Causes, International Management Review, 93-106, 5(1), 2009

[5] The CHAOS Manifesto, 2013: Think Big, Act Small by The Standish Group International, 2013.

[6] Rupinder Kaur, Dr. Jyotsna Sengupta, Software Process Models and Analysis on Failure of Software Development Projects, International Journal of Scientific & Engineering Research, 1-4, 2(2), 2011

[7] Fabriek, Matthias, Elt. Al, Reasons For Success And Failure In Offshore Software Development Projects, [Available: http://is2.lse.ac.uk/asp/aspecis/20080039.pdf], [Accessed on: 05/09/2014]

[8] Alpha Software Inc, Why Software Projects Fail: A New Assessment of Risk, [Available at http://www.alphasoftware.com/documentation/200801_wpaper/low_risk_a5.pdf], [Accessed on: 05/01/2015]

[9] Lorin J. May, Major Causes of Software Project Failures. [Available at] http://www.cic.unb.br/~genaina/ES/ManMonth/SoftwareProjectFailures.pdf [Accessed on: 10.12.14]

[10] Khaled El Emam and A. Günes¸ Koru, A Replicated Survey of IT Software Project Failures, IEEE Software, 84-90, 2008.

[11] Watts S. Humphrey, Why Big Software Projects Fail: The 12 Key Questions, CROSSTALK: The Journal of Defense Software Engineering, 25-29, 18(3), March 2005

[12] Watts S. Humphrey, Five reasons why software projects fail, [Available at: http://www.computerworld.com/s/article/71209/Why_Projects_Fail?taxonomyId=11&pageNumber=2] [Accessed on: 15/12/14]

[13] By Tom Gilb. Project Failure: Some Causes and Cures, Edited MASTER paper, Version: February 29, 2004

[14] Capers Jones, Social and Technical Reasons for Software Project Failures, CROSSTALK: The Journal of Defense Software Engineering, 2-9, 19(6), June 2006

[15] Mandal A., Pal S. C., Investigating and Analysing the Desired Characteristics of Software Development Lifecycle (Sdlc) Models, International Journal Of Software Engineering Research & Practices, 9-15, 2(4), 2012.

[16] Mandal A., Pal S. C., Achieving agility through BRIDGE process model: an approach to integrate the agile and disciplined software development, Innovations in Systems and Software Engineering: A NASA Journal, 1-7, 11(1), [DOI 10.1007/s11334-014-0239-x], 2015

[17] Mandal A., Pal S. C., Emergence of Component Based Software Engineering, International Journal of Advanced Research in Computer Science and Software Engineering, 311-315, 2(3), 2012

[18] Mandal A., Pal S. C., A Comparative Analysis of BRIDGE and Some other Well Known Software Development Life Cycle Models, International Journal of Computer Science & Engineering Technology (IJCSET), 196-202, 5(3), 2014.

A Comparative Analysis of BRIDGE and Some Other Well Known Software Development Life Cycle Models

Ardhendu Mandal*

Department of Computer Science and Application, University of North Bengal, Raja Rammohunpur, P.O.- N.B.U., Dist-Darjeeling, State-West Bengal, Pin-734013, India.
am.csa.nbu@gmail.com

S. C. Pal

Department of Computer Science and Application, University of North Bengal, Raja Rammohunpur, P.O.- N.B.U., Dist-Darjeeling, State-West Bengal, Pin-734013, India.
schpal@rediffmail.com

Abstract— The existing Software Development Life Cycle Models (SDLC) models were quite successful earlier, but are rarely used in modern software development because of their limitations and non suitability for modern projects. To cater with the present software crisis, Mandal [13] proposed a SDLC model named BRIDGE for modern software development. In this paper we have outlined the BRIDGE process model. Further we performed a comparative analysis of the existing well know models and BRIDGE. Then, we discussed the results of the comparative analysis. Finally we conclude by recommending the BRIDGE process model to be the best generic process model for software development suitable for modern software development projects.

Keywords- Software Development Process Model (SDLC), BRIDGE Process Model, Comparative Analysis.

I. INTRODUCTION

The rapid development in the hardware technology has made modern processors very efficient and powerful. Hence, the expectations from the software have gone to zenith. But the complexity of the modern software are much complex as compare to those of earlier. Development cost, time and quality of the modern software are in crisis. There are several Software Development Life Cycle (SDLC) Models i.e. Classical Waterfall, Spiral, Prototype, V-Model, evolutionary model etc. All these SDLC models have several advantages as well as some limitations. A software (SW) project, irrespective of its size, goes through certain defined stages, which together, are known as the Software Development Life Cycle (SDLC). Life Cycle refers to the different phases involved starting from the project initiation to project retirement. For better understanding and implementation of the various phases of software development, different software development models have been developed and proposed so far. A few well known models are waterfall model, spiral model, evolutionary model, prototype model, V model etc. It is pre established that different SDLC models have different capabilities and limitations. Hence, selecting suitable SDLC model for any project is quite crucial as not all process models are good for any type of project. Hence, analyzing the different SDLC model is significant and helps one to select the appropriate model for a project. Recently a few more new process models are proposed with the well known traditional models to accommodate the new industrial needs.

II. SOFTWARE DEVELOPMENT APPROACH, PROCESS AND PROCESS MODEL

It is really tough to draw a sharp line between software development approaches and SDLC process models. In many literature of software engineering, these terms are used interchangeably or confusedly. So, before we begin the details discussion of the topic, let us somehow draw the boundary line between software development approaches and SDLC process models. Defining these two terms are beyond the scope of this paper. Here we just try to explain both only to establish the differences from our point of view. SW development process or simply process typically defines the set steps to be carried out during the development of the system. SW development life cycle (SDLC) is the time from the concept development to the product retirement i.e. the time of SW process. SW development life cycle (SDLC) process model typically depicts the fashions in which the SW process to be carried out i.e. which steps/phases to be done before or after another step/phase. In general all the process models do cover all distinct phases defined by SW process, but in different manner or sequence- which makes one process model differ from the other. In other words, a software development process model is an approach to the Software Development Life Cycle (SDLC) that describes the sequence of steps to be followed while developing software projects [10, 18]. We consider Agile, incremental, extreme and iterative as approach or philosophy to software development rather than as process model which can be implemented following other process models i.e. Waterfall, RAD, Spiral, Prototyping or alike.

III. DIFFERENT WELL KNOWN SDLC PROCESS MODELS

Many people have proposed different software development process models. Many are quite same in different aspects while other differs. Here we just consider some well known SDLC process models enlisted below:

Waterfall Model: It is a software development model with strictly one Iteration/phase. In this process model, development proceeds sequentially through the phases: requirements analysis, design, coding, testing, integration, and maintenance [23].

Evolutionary: Evolutionary development uses small, incremental product releases, frequent delivery to users, dynamic plans and processes. The evolutionary development model divides the development cycle into smaller, incremental waterfall models in which users are able to get access to the product at the end of each cycle. The users provide feedback on the product for the planning stage of the next cycle and the development team responds accordingly by changing the product, plans, process etc [7, 17].

Prototype Model: It is a software development process that begins with requirements collection, followed by prototyping and user evaluation. This model facilitates to discover new or hidden requirements during the development [8].

Spiral Model: This process model proposes incremental development, using the waterfall model for each step, with more emphasis on managing risk [3].

V-Model: This is an extension of the waterfall model which emphasizes parallelism of activities of construction and verification. Here, the process steps instead of moving down in a linear way bend upwards after the coding phase resulting in the typical V shape formation.

RAD Model: It is a software development process that allows usable systems to be built in as little as 60-90 days, often with some compromises.

The details discussion of these SDLC model is beyond the scope of this paper, but just highlight the features of these models which are important for considerations. The readers may follow the references for further detail discussion of these process models [16, 21, 15]. In the following section we just briefly explain the SDLC model BRIDGE which is our prime concern.

IV. BRIDGE PROCESS MODEL IN A NUTSHELL

Although the details discussion of the BRIDGE model is beyond the scope of this paper, just the schematic diagram of the BRIDGE process model is given below in *Figure 1* with its analytical results.

The in-depth study of the BRIDGE model discloses a lot of information that may be used to analyze the model. These are briefly discussed below [13]:

- It involves the client over the entire development life cycle activities.
- It keeps continuous communication with the project management team.
- It explicit verification of individual phases.
- Separate software architecture design phase.
- Separate system deployment phase.
- Separate on-site system testing phase.
- Supports components based software development.
- It emphasizes on standard coding.
- It considers configuration management as a separate activity.
- It forces to specify all the phase deliverables.
- It explicitly instructs to validate the system.

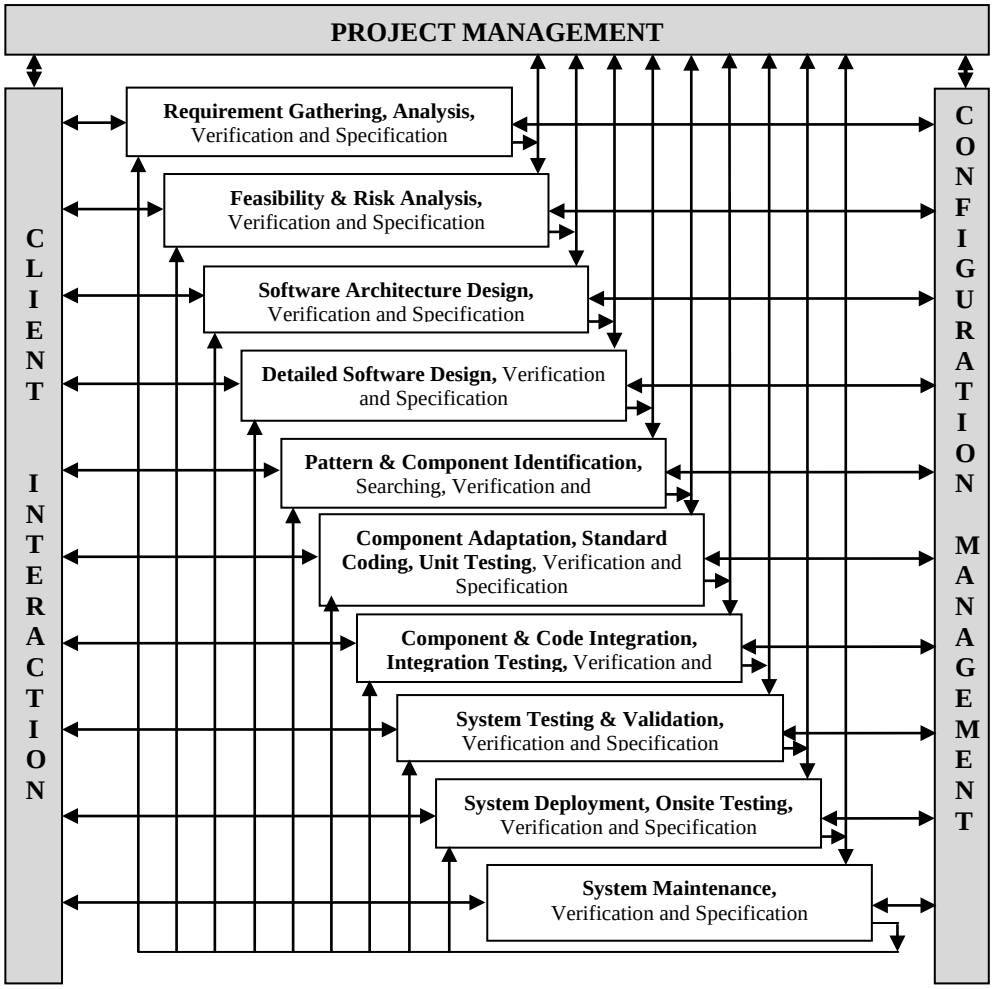


Figure 1. BRIDGE Process Model [13]

V. PARAMETER SELECTION FOR COMPARATIVE ANALYSIS

Below we enlist and discuss briefly the parameters alphabetically those we have considered for the purpose of comparative analysis:

Adaptability: This is the ability to react to operational changes as the project is developed. Change orders are easily assimilated without undue project delay and cost increases.

Budget: Budget remains one of the most significant crisis for software development projects. Some process model like Spiral and Prototyping increases the project cost as compared to others. Hence a SDLC process model has great impact on software development cost or budget.

Changes Incorporated: Change is unavoidable in software development. Managing change is a critical component of any SDLC model. Change Management and SLDC are not mutually exclusive. Change management occurs throughout the development life cycles which need to be incorporated in the system development.

Complexity of the SDLC: Different SDLC process model have varying degree of complexity. Some are easy to use and implement while others are not.

Documentation: Documentation of software development process is very important but time consuming and expensive. To reduce development time and cost, agile philosophy recommends less document which remains one of the most important critic of agile philosophy. Documentation plays vital roles in system development, implementation, maintenance and project management. But, not all process models facilitate and recommend adequate and sufficient documentation.

Expertise Required: Although some process models are better over others, but need some kind of expertise during its use and implementations in various phases at varying degrees. To avail the advantages of some process models i.e. Spiral, BRIDGE [13] and others which supports reusability- the software engineers required certain level of expertise.

Flexibility: The freedom afforded to software architects, analysts or developers to tailor the software development process according to business needs and project characteristics is a crucial factor in successful project completion. The software development organization often can benefit from introducing flexibility into their software development methodologies [20].

Guarantee of Success: This is really crucial to measure whether any process model will guarantee success or not. If so, up to what extent wills the process model guarantees the success is a big question need to be explored. As the project success depends upon many other constraints and parameters, but given the other parameters as desired, project success may vary from following one SDLC model to another.

Integrity & Security: Including security early in the system development life cycle (SDLC) will usually result in less expensive and more effective security than adding it to an operational system. To be most effective, information security must be integrated into the SDLC from its inception [9].

Maintenance: Systems are dynamic and the model offers the ability to produce a final project that is inherently designed for maintenance. This includes such items as cumulative documentation.

Management Control: Management will have the ability to redirect and if necessary redefine the project once it is begun. A key phrase is 'incremental management control', with each step under tight management control. Management control has great impact on project success.

Overlapping Phases: Each step of the project is to be completed before another is begun. Project modules are distinct and easily identifiable.

Parallel development: Parallel development support, if possible to employ may increase productivity and reduce development time while optimally utilizing the resources.

Productivity: The SDLC must ensure that the expected return on investment (ROI) for each project is well defined. The SDLC must minimize the unnecessary rework. It must be designed in such a way as to take maximum advantage of the computer assisted software engineering (CASE) tools. At the same time the SDLC must utilize the resources most effectively and efficiently to improve the productivity.

Progress Measurement: Progress measurement allows development team as well as the project management team to determine how well tasks were estimated, how well they were defined, and whether items are completed on-time and within-budget. Any SDLC process model should provide the facility to measure the progress during the system development.

Quality Control: Each module of the project can be thoroughly tested before another module is begun. Project requirements are measured against actual results. Milestones and deliverables can be used for each step of the project.

Requirements Specification: Depending on the project nature, the requirement may be identified at the very beginning of the project development or may be discovered during the development process. But, not all process model supports requirement discovery over the development process. Hence, requirement specification may be static or may be dynamic. Any SDLC process model should take into account the issue of requirement specification.

Requirements Understanding: Some process model needs the requirement must be well understood before the development process stated, while other may allow understanding the requirements over the development process. One may start with the initial understanding of the requirement and during the development the requirement understanding increases gradually.

Reusability: Reusability is one of the most significant and efficient attribute of any SDLC process model these days. Reusability helps to improve system productivity, reduce cost and system delivery on-time. The degree of reusability support may vary from one process model to another.

Risk Involvement: The risk involvement may vary from one model to another depending on the nature of requirement understanding capability support by the process model. Apart from this, there may be several other sources of risks involvement.

Risk Management: Different types of risks are implicit part of any project. Levels of risk are identifiable and assessment strategies available. Strategies are proved for over-all and unit risks.

Table: 1 Comparative Analysis

Models / Parameters	BRIDGE	Waterfall	Prototype	Evolutionary	Spiral	RAD	V-Shape
Adaptable	Excellent	Limited	Good	Good	Excellent	Limited	Limited
Budget	Low	Low	High	Low	High	Low	Low
Changes Incorporate	Easy	Impossible / Difficult	Easy	Easy	Easy	Easy	Difficult
Complexity	Medium	Simple	Moderate	Complex	Complex	Medium	Simple
Documentation	Yes	Strong	Weak	Moderate	Moderate	Poor/ Limited	Yes
Expertise Required	Medium	Low	Low	Low	High	Medium	Medium
Flexibility	Flexible	Inflexible	Highly Flexible	Highly Flexible	Flexible	High	Rigid
Guarantee of Success	High	Less	Good	Good	High	Good	High
Integrity and Security	High	Vital	Weak	Weak	High	Vital	Limited
Maintenance	Easily Maintained	Least Glamorous	Routine Maintenance	May be overlooked	Typical	Easily maintained	Lest
Management Control	Yes, Dedicated	No	No	Weak	Moderate	Weak	Weak
Overlapping Phases	May be	No	Yes	Yes	Yes	No	No
Parallel Development	Supported	No	No	Limited	Limited	No	Limited
Productivity	Highest	High	Improved	Improved	High	Improved	Improved
Progress Measurement	Measurable	Easily Monitored	Measurable	Measurable	Measurable	Measurable	Measurable
Quality Control	Very Good	Poor	Moderate	Good	Good	Adequate	Moderate
Requirements Specification	Adaptable /Dynamic	At the Beginning	Frequently Changed	Frequently Changed	At the Beginning	Time-box Release	At the Beginning
Requirements Understanding	Well Understood	Well Understood	Not Well Understood	Not Well Understood	Well Understood	Easily Understood	Easily Understood
Reusability	Excellent	Limited	Poor	Poor	Moderate	Moderate	Moderate
Risk Involvement	Low	High	Low	Moderate	Low	Little	Low
Risk Management	Highly Supporter	Not Considered	Moderate	Good	Highly Supporter	Poor	No
Simplicity	Intermediate	Simple	Simple	Intermediate	Intermediate	Very Simple	Simple
System Delivery	Early and periodic partial operational system	At the end of the system development	At the end of the system development	Early and periodic partial operational system	At the end of the system development	At the end of the system development	At the end of the system development
Time	Shortest	Short	Long	Long	Long	Short	Short
Understandability and Implementation	Moderate	Easy	Easy	Moderate	Complex	Moderate	Easy
User Involvement	Throughout Process	At the beginning	High/Up to design phases	Throughout Process	High	Throughout Process	At the beginning

Simplicity: Any model need is easy to understand and to implement. Simplicity of any process model reduces the burden of expertise and improves productivity while reduce development cost and project risk.

System Delivery: The system may be delivered either partially as individual operational module wise or as the complete system with full functionality at once.

Time: Time is actually referred to as Time Horizon because we are interested in knowing the projected completion of the project. The development time may vary from one process to another.

Understandability and Implementation: Different process model may need varying level of expertise. Simple and better understandable process model are always easy to implement.

User Involvement: Any model lends itself to strong and constant end-user involvement. This includes project design as well as interaction during all phases of project development.

VI. COMPARATIVE ANALYSIS

The comparisons among different SDLC models in respect to the features discussed above are illustrated in *Table 1* [2, 12, 11, 6, 19, 22, 5, 14, 1, 4]. From the above comparative analysis, it is established that the BRIDGE process model possesses many suitable features in comparison to the other process model.

VII. CONCLUSION

There exist several well known SDLC process models. One process model has different comparative advantages from the others in many respects. But no process model is just good for any type of project. So it is not blindly recommended to choose any process model for any project! The above comparative study shows that overall the BRIDGE process model has several competitive advantages over the other existing well known process models. As BRIDGE model has excellent adaptability, supports process tailoring and other attributes, we recommend this SDLC process model to be used for any types of software development projects.

VIII. FUTURE WORK

In near future we would like to validate the result of this theoretical comparative analysis by means of practical experimental statistical results. We are implementing several instances of one sample project following BEIDGE and different other models individually by different teams to perform practical experimental comparative analysis. During the experimental we shall refine the BRIDGE model if necessary to make this model the best alternative among the others. Further, we are working to explore the different ways to achieve the agile philosophy following BRIDGE process model.

REFERENCES

[1] Alexander L. and Davis A., Criteria for Selecting Software Process Models, presented at COMPSAC, 1991.
[2] Ali M.N. M. and Govardhan A., A Comparison Between Five Models Of Software Engineering, International Journal of Computer Science Issues, 7(5), 2010.
[3] Boehm B. W., A Spiral Model of Software Development and Enhancement, IEEE Computer, 21(5), pp. 61-72, 1988.
[4] Comer, E., Alternative Software Life Cycle Models, Proc. of International Conference on Software Engineering, 1997.
[5] Davis, A, Bersoff, E, Comer, E, A Strategy for Comparing Alternative Software Development Life Cycle Models, IEEE Transactions on Software Engineering, 14(10), pp. 1453-1461, 1988.
[6] Dholakia P. and Mankad D., The Comparative Research on Various Software Development Process Model, International Journal of Scientific and Research Publications, 3(3), 2013.
[7] Elaine L. May and Barbara A. Zimmer, The Evolutionary Development Model for Software, Hewlett-Packard Journal, 1996.
[8] Gomma H. and Scott D. B. H., Prototyping as a tool in the specification of user requirements. Proc. of Fifth Int. Conf. on Software Engineering, pp. 333-341, 1981.
[9] Grance T., Hash J. and Stevens M., Security Considerations in the Information System Development Life Cycle, NIST SPECIAL PUBLICATION 800-64 REV. 1, 2003.
[10] Guimares L. and Vilela P., Comparing Software Development Models Using CDM, Proceedings of The 6th Conference on Information Technology Education, New Jersey, pp. 339-347, 2005.
[11] Hijazi H., Khdour T. and Alarabeyyat A., A Review of Risk Management in Different Software Development Methodologies, International Journal of Computer Applications, 45(7), 2012.
[12] Malhotra S. and Malhotra S., Analysis and tabular comparison of popular SDLC models, International Journal of Advances in Computing and Information Technology, 1(3),2012
[13] Mandal A., BRIDGE: A Model for Modern Software Development Process to Cater the Present Software Crisis, Proc. IEEE Int'l Conf. Advance Computing Conference, pp. 494-500, 2009. Also available at IEEEXplore with DOI: 10.1109/ IADCC.2009.4809259
[14] Molokken-Ostvold J. and Jorgensen M., A comparison of software project overruns - flexible versus sequential development models, IEEE Transactions on Software Engineering, 31(9), pp. 754-766, 2005.
[15] Pfleeger S. L. and Atlee J. M., Software Engineering: Theory and Practice, Pearson, 2011.
[16] Pressman R. S., Software Engineering: A Practitioner's Approach, MaGrawHill Publications, 2005.
[17] Rlewallen, Software Development Life Cycle Models, 2005, http://codebeter.com.
[18] Ruparelia N., Software Development Lifecycle Models, ACM SIGSOFT Software Engineering Notes, 35(3), pp. 8-13, 2010.
[19] Sasankar B. A. and Chavan V., Survey of Software Life Cycle Models by Various Documented Standards, International Journal of Computer Science and Technology, 2(4), 2011.

[20] Sharad Chandak S., Rangarajan V., Flexibility in Software Development Methodologies: Needs and Benefits (Executive Summary), http://www.cognizant.com/InsightsWhitepapers/Flexibility-in-Software-Development-Methodologies-Needs-and-Benefits.pdf
[21] Sommerville I., Software Engineering, Pearson Education, 8th Edition, 2009.
[22] Taya S. and Gupta S., Comparative Analysis of Software Development Life Cycle Models, International Journal of Computer Science and Technology, 2(4), 2011.
[23] Walker W. Royce. Managing the development of large software systems: concepts and techniques, Proc. IEEE WESTCON, Los Angeles (August 1970) Reprinted in the Proceedings of the Ninth International Conference on Software Engineering, pp.328-338, 1987.

Investigating and Analysing the Desired Characteristics of Software Development Lifecycle (SDLC) Models

Ardhendu Mandal and S. C. Pal

Abstract—There exist several Software Development Lifecycle (SDLC) Models, but they are rarely followed by organizations for the real project implementations as they lack suitability. On the way to investigate the reasons for non suitability of these models, it is exposed that there are insufficient parameters and metric for judging the characteristics of any SDLC model. In this paper, we have investigated, identified, enlisted and analyzed the different parameters of SDLC process models. The result of this work is of great significance as these results i) may help while developing any new SDLC model ii) may even help the development team to choose the best suitable model among the alternatives for any project and iii) the outcome of this work may further be used to design and develop metrics for SDLC characterization.

Index Terms—Software Engineering, Software Development Lifecycle (SDLC), Process Model, Characterization.

1 INTRODUCTION

SEVERAL Software Development Lifecycle Models (SDLCs) are in existence [1], [2], [3]. Over the time different people have proposed different models to meet the industrial demands. Any SDLC process model should be a repeatable, clearly documented, highly-effective and must be based on the best industrial practices. However, the traditional SDLC process models provide very insightful theory and helpful best practices, but do not provide the practical details for daily application. As a result, statistics [4] shows that SDLC process models are rarely used by organizations for the purpose they are designed and developed for. Another primary reason for not using these models is due to lack of their suitability for real life projects - which led to software crisis. While investigating the reasons for unsuitability of these models, it is identified that we lack well defined characteristic parameters for any SDLC model. Without applying the process model in real project, we do not have adequate metric to analyze the suitability and goodness of such models. Davis et al [5] has proposed a strategy long back in 1988 for comparing alternative SDLCs only based on the ability to satisfy user needs and reduced life cycle cost. In this work, we have investigated, identified and analyzed the features of any SDLC process model in general which may further be used for characterizing any SDLC process model for its suitability.

2 OBJECTIVES AND GOAL

The objective of this work is to identify the different characteristics of a good software development lifecycle model. Given these characteristics, one can judge, evaluate, predict and select the best SDLC model suitable for real projects. The outcome of this work then can be used while designing and developing new process models too. Using such metric, one may evaluate a model for its suitability, applicability and predictability of success for any project. Moreover, these can be used to design the quality, suitability and predictability metric of any process model. The outcome of this research may further be used to develop new SDLC models according to the need of the industry and even may be used while designing any new process model. Hence, the goal of this work is to develop the foundations for SDLC metric.

We shall use the term software development lifecycle process model and process model over the paper interchangeably.

3 SDLC PROCESS MODELS AND OBJECTIVES

There are numerous examples of disasters that had been caused by software failures. As the computerization of the society continues, the public risks of poor quality software will become untenable unless orderly steps are taken to improve the software processes [6].

Any SDLC process model has three primary business objectives [7]:

- Ensure the delivery of high quality systems,
- Provide strong management controls over the projects, and
- Maximize the productivity of the systems development team.

Further, these objectives can be broadly categorized from the following two perspectives:

a) The Technical Perspectives

While building a system, there remain many technical activities and issues including system definition (analysis, design, coding), testing, system installation (e.g.,

Ardhendu Mandal is with the Department of Computer Science and Application, University of North Bengal, Darjeeling, West Bengal, India. E-mail: am.csa.nbu@gmail.com

S. C. Pal is with the Department of Computer Science and Application, University of North Bengal, Darjeeling, West Bengal, India. E-mail: schpal@rediffmail.com

training, data conversion), production support (e.g., problem management), defining releases, evaluating alternatives, reconciling information across phases and to a global view and defining the project's technical strategy etc. to be resolved.

b) The Management Perspectives

When we plan to develop, acquire or revise a system, we must be absolutely clear with the objectives of that system. The objectives must be stated in terms of the expected benefits that the business expects from investing in that system. The objectives should exhibit the expected return on investments. To achieve the project objectives and goal many management related issues have to addressed and resolved. The primary management activities includes setting priorities, defining objectives, project tracking and status reporting, change control, risk assessment, step-wise commitment, cost/ benefit analysis, user interaction, managing vendors, post implementation reviews, and quality assurance reviews etc.

All the above objectives irrespective of technical or managerial, has to be achieved through some SDLC process model if possible. But, unfortunately not all the available process model does address these issues efficiently.

4 Desired Characteristics of SDLC Models

In order to meet the project objectives and goal, SDLC have to satisfy many specific requirements i.e. being able to support different types of projects and systems of varying scopes, supporting both the technical and management activities, being highly usable, and providing guidance on how to execute and install it for solving real life problems and many more. In the following section we are going to identify, enlist and discuss briefly some primary characteristics that are expected from any SDLC process model. As the degree of importance of these characteristics does vary from project to project, here we just enlist these characteristics alphabetically.

Change Management

Requirement changes are often necessary, frequent and inevitable. The drivers of requirement changes may be customer demand, technical demand, competitive demand or even governmental or business policy demand. While occasional changes are essential, historical evidence demonstrates that the vast bulk of changes can be deferred and phased in at a subsequent point. To develop quality software on a predictable schedule, the requirements must be established and maintained with reasonable stability throughout the development cycle. Changes will have to be made, but they must be managed and introduced in an orderly way. Hence, change management is a critical part of any SDLC model. As requirements are changed frequently, there is a need of streamlined flexible approach to manage these requirement changes within the SDLC model. Although requirement change management and SLDC are not mutually exclusive but the change management activities occurs throughout the development process. Further, cost of adopting changes is higher after the completion of the development. Hence, the objective should be to limit the change management activities within the initial development period as much as possible. If change is not controlled, orderly testing is impossible and no quality plan can be effective.

Concurrent and Parallel Development

If high cohesion and low coupling modular system design is possible, then concurrent, distributed and parallel system development activities can be employed. This can improve productivity, timely system delivery while reducing total development cost and optimal usage of available resources.

Coordination among project stake holders

A software project is not an individuals' job, but a collective effort towards the common goal. The success of software development projects depends on carefully coordinating the effort of many individuals across the multiple stages of the development process. Coordination—long recognized as one of the fundamental problems of software engineering —has become ever more challenging. This has led to a growing body of work on coordination in software development [8]. With the rapid advancement of Information and Communication Technilogy (ICT), the location or centrer specific project evelopment barrier are being diminishing. For optimal resource utilization, often project development activities are distributed over different development centres. Further, more the people are involved in one job, more the chances of misunderstanding and communication gap. Hence, if large numbers of people are involved and scattered over different development centres, the process model must provide mechanism for better coordination among the project stakeholders.

Cost of Life Cycle Implementation

Additional costs and overheads are the primary barrier in process model implemention. For this reasons, many organizations do not implement or follow any process model. An ideal processs model implementation should be economic, easy and justifiable. It should not require additional, special application or software purchase to effectively perform the process implementation or ongoing process management. In addition, it shoud be easily automated utilizing any internal process management software tools currently being used within an organization.

Customer Involvement and Interaction

In most of the common process model, there is no direct communication among customer, development team or project management team throutout the development

process. In traditional models, management plays the vital role of bipartite body who works as the communication channel and messenger in the communication between customer and development team. As a result always there remains some communication gap and some missing or hidden information yet to convey to the development team but with the management. As a result, often proper requirements remain unspoken or hidden to the development team. Even conveyance of information might cause a loss of knowledge, as great amount of data remains with its carrier and never get handed off to others [9]. Some interviewees suggested that the lack of direct contact between the development team and the customers could encumber the process of specifying requirements for the future. In turn handoffs among functions can cause delays and increasing risks of information being misunderstood [10]. According to interviewees, level of details is varying depending on representatives between customer and developer. As a result, the developed system is frequently not satisfactory or even lead to project failure. Allowing direct communication of development team with the customer during the entire development process could eliminate project completion time and recourses consuming non value- added-action in form of handoffs, therefore waste [11]. Hence, user or customer involvement during all phases of the project development is very important for project success and must be supported by any process model.

Proper and Sufficient Documentation

Documentation has two ways of influencing the development process. Firstly, it makes the development process easier to understand. Secondly, documentation enables easy system maintenance, which can be linked to one of the principles in lean philosophy – knowledge sharing. But, unnecessary documentation can be addressed as waste. Any process model must enforce to develop the necessary document concurrently with the development pocess, while must avoid producing unnecessary documents to prevent miss utilization or waste of resources as in the case of agile development philosophy.

Early Defect Removal

In case of any error or defect, if possible, it is always better to remove or rectify them in the earliest phases of the SDLC process model. Hence, the process model must focus on identifying the errors in the same or closest phase of the SDLC process to avoid or reduce the redo-work and cost. The best way to identify errors is to perform a close and effective verification after each and every phase and to set specific predefined phase entry and phase exit criteria effectively.

Easy to Execute

Not all process models are easy to execute. Some process execution may require additional focus than the other. But often degree of easiness in execution may affect the other evaluation criteria of the process model. Always neither all easy process models are bad nor are all complex process models good. Hence, the tradeoff must be resolved depending on the suitability project demand.

Effective Management and Control

Most of the existing traditional SDLC process models don't involve management team directly with the development team. Hence, the project management team does not have direct communication with the development and associated members. The management just remains as a silent intermediate communication body. Thus, proper management observation and control is hidden in the development process. As a result, the development process lacks proper management supervision and controls. In addition, the project has to suffer from resource shortage, risk handling, coordination and many other conflicts and problems. To overcome these problems, a direct involvement of project management team with the development team is necessary and important. The software development process must be under statistical control of the project management team to produce consistent and better result through process improvment.

Focus Towards Goal

The project objectives and goal must be well defined and specific. The process model should view software development within the context of the larger system level definition, design, and development. Further, it should recognize both the potential value of opportunity and the potential impact of adverse effects, such as cost overrun, time delay or project risks according to the system and project specifications. Hence, any process model must reflect the project scope, objectives and goal consistently during the development process.

Incremental

Software development project may be divided into distinguishable cascading phases. Before starting a phase, it may require a defined set of inputs from its immediate previous phase. The incremental methodology maintains a series of such phases. However, in the design phase development is broken into a series of increments that can be implemented sequentially or in parallel. The subsequent phases do not change the requirements rather build upon them towards the project completion. The methodology continues focusing only on achieving the subset of requirements for that development increment and continues all the way through implementation. Increments can be discrete components, functionalities, or even integration activities. Hence, through the incremental methodology, quantifiable partial solutions may be given to the customer without waiting for the entire project to be completed. The subsequent partial system may be developed in parallel to the already de-

veloped and operational partial system that may be further integrated when the second incremental development is available. The incremental process increases the degree of customer satisfaction and product quality as the defect of the delivered partial system may be identified while at operation and necessary changes may be incorporated immediately and deliver with the next increment.

Iterative

In iterative process, the development begins by specifying and implementing the partial software requirements available at the moment without waiting for the complete or full software requirements specification. Further, these partial requirements can be reviewed in order to identify additional requirements and necessary modifications are made. This process is then repeated to implement the newly identified and specified requirements producing a new version of the software at each such iteration of the process model. This allows the project team to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the software in use. Iteration enhances the ability of the project management team to efficiently address the requirements of stakeholders and to complete, review, and revises phase activities until they produce satisfactory results. The product is defined as completed when it satisfies all of its requirements.

Distributed or Multi-Site Software Development

Recent advances in information technology have made Internet-based collaboration much easier. It is now possible for a software team to draw on talented developers from around the world without the need to gather them together physically. To solve the problems like team relocation and project delay, developing software at multiple sites has been considered these days. Besides the obvious advantage of being able to tap into a much larger pool of human resources, experts working together from two or more locations can actually yield better outcomes [12]. In such cases, software managers have to be able to manage these distributed teams. They need to define sharper processes, tracked, overseen and ensure that they are followed.

Quick Implementation

A predefined solution that does not require organizations to start from the very initial level would allow organizations to exponentially cut down the time required to fully complete a process implementation effort. A process blueprint solution would drastically reduce the time and cost associated with traditional process improvement by laying an effective foundation for SDLC organizations to build upon.

Phase Length and Cycle Duration

Phase length and Cycle duration should be optimal. It is well known that long cycle and phase duration is one of the primary reasons for project failure. Further, long phase and cycle duration makes the performance measurement task more difficult. If the cycle and phase duration is long, there may be problem with resource scheduling and may promote un-optimized utilization of resources which may affect the project cost, quality and schedule. Another source of risk resides in the relatively long stages, which makes it difficult to estimate time, cost and other required resources for project completion. Further, if the cycle duration is more, the delivery of incremental and partial system delivery will get delayed that may decrease customer satisfaction. Hence, the process model must be designed in such a way that the duration of each cycle and length of each phase must be small.

Predictability

Any process model should be predictable. That is, cost estimates, schedule and quality commitments would be met with reasonable consistency, and the quality of the resulting products would generally meet the users' needs [9]. As money, time, people and many other resources are involved, and at the same time quality and customer satisfaction are prime concern, before starting the project we need to predict the future outcome from it. If the outcome is not favorable, carrying out the project is nothing but waste of resources and gaining loss! In addition, the process model should ensure that we can produce desired functions with higher quality using optimal resources in lesser time in a predictable manner. Thus, the process should have a predefined level of precision to facilitate a complete, correct and predictable solution.

Process Tailoring

There may be different kind of projects and situations when no standard process is applicable. In such cases, the process model should provide the flexibility to adopt itself with the project and situational demand maintaining the integrity and consistency of the process by permitting tailoring of the standard process. Hence, tailoring is the process of adjusting the standard process to obtain a process that is suitable for the project need and situation [13]. Thus, process tailoring facility makes the process model more flexible and adaptable.

Progress Measurement

To evaluate the project performance and management control progress measurement of the project work is very important. For this purpose, milestones need to be set at regular intervals. Otherwise the project may suffer from 99% complete syndrome [1], [2]. Effective and proper project measurement is only possible when the development process is under statistical control [14].

Prototyping

Prototyping is necessary when it is very difficult to obtain exact requirements from the customer at the begin-

ning of the project. Given the prototype of the system, user keeps giving feedbacks from time to time and according to the feedbacknecessary modifications are incorporated in the proposed or tobe developed system. By doing these, the hidden, unidentified user requirements may be discovered during the initial phases of the system development process. By doing so, project failure risks may br reduces while improving the the degree of user satisfaction and system quality.

Quality Control

Lack of quality assurance during the different phases of the development process is often a potential source of risk. Validating the product is restricted to a single testing phase lately in the development process. Hence, the testing phase is the highest risky phase, since it is the last stage wherein the system is put as a subject for testing. Thus, all problems, bugs, and risks are discovered too late when the recovering from these problems requires large rework which consumes time, cost, and effort. Milestones and deliverables can be setup or specified for each step of the project. Each module of the project is thoroughly tested before the beginning of another module. Project requirements are measured against the actual results.

Reliability

Any process model must ensure development of quality reliable systems. A potential source of risk resides in the relatively long stages of any process podel, which makes it difficult to estimate, time, cost, and other resources required to complete each stage successfully. In general, if incremental model is followed, the partial working systems are delivered periodically. It increases the reliability of the process model as reliability of the system does increase gradually during each partial product delivery to the customer.

Repeatable

A SDLC process model should be repeatable i.e. the process should be repeat in case of projects which are similar in type or belongs to similar domain. A repeatable process reduces the cost of process model implementation as it is well known, learned and experienced to the development team. Hence, repeat process will reduce the project risks and cost. The quality of the system will be better as the outcomes of the phases are predictable.

Reuse

The advantages of reuse in developments are now well established. Studies show that reuse had great impact on productivity, cost, quality, time-to-market and customer satisfaction [15]. But very few process models like BDIDGE [4] are designed keeping the view of reusability in mind. Hence, as reuse has potential advantages, support of reusability is an important desired characteristic for process model in recent days.

Risk Management

Risk is commonly defined as a measure of the probability and severity of adverse effects [16]. Risks are an inherent part of any project which must be managed in advanced (if possible) or during the development process. As all projects inherit some risks, it is desired that the process model should provide adequate scope for risk management. Project risk may be related to project staffing, resources, schedules/ budgets, technical, requirements changes etc. The SDLC model must focus on the risk associated with the project continuously so that the management can take necessary control measure to prevent project failure. To manage risks, any process model must provide direct control over the project by project management. But, it has been seen that a very few model i.e. spiral and BRIDGE [4] provides such direct management support to the development process.

Scope

Client demands never ends! The more you provide, the more they demand! Hence, if the scope of the project is undefined, satisfying customer is just a dream. The process should clearly mention what is desired to be produced and the developed product should be comparable to the defined requirements. Thus, the process model should have its specific scope as well as must limit the scope of the project. Scope of the process model bounds the suitability of the process model for different types of systems and projects.

Security Assurance

Effective security is incorporated at the onset of a project. If it is included as a requirement early in the system development and/ or acquisition process, it typically results in less expensive and more cost effective security. Waiting to integrate security until later in the process usually results in interoperability issues and increased cost. Integrating security into the SDLC begins with being able to articulate the security properties desired within the system. This process is typically cyclical in refinement beginning at the top level and drilling down into what will eventually be security specifications. There are many ways to express the high-level security requirements i.e. ISO 15408 and others.

Separation of Concern in Different Phases

As the development process is divided into different phases with distinct objectives, hence the concern of different phases should be clear-cut and well separated. Otherwise, there may be chaos during progress measurement, quality and other management problems. Without separation of concerns in different phases, it will be tough to set milestones effectivel, inturn that will make the progress measurement task difficult.

Simplicity and Flexibility

Neither all simple things are better, nor are all complex things bad! The process model should be simple to un-

derstand, easy to follow and well manageable. More the process model flexible, it is easier to manage and follow for execution. As change is inevitable due to specific nature of system projects, flexibility to accommodate changes is a basic need from a process model.

Software Process Improvement and Feedback

Software Process Improvement (SPI) is an approach to designing and defining new and improved software process to achieve basic business goals and objectives i.e. increase revenue or profit, and to decrease operating costs by manipulating or changing the software process. The objectives of software process improvement (SPI) are to process produce products according to plan while simultaneously improving the organization's capability to produce better products [17]. Perfection of any process is done via constant improvements. During the project execution or at the end, the team members give feedback in the form of reports suggesting different ways to improve the development process for the next iterations or future. One of the conditions required to improve the development process is to have the input from previous cycles so that the team's opinion and experience gathered during previous phase can be dissiminated among other teams. This supports the process improvement throughout the whole organization, by adopting one of the core lean principles such as knowledge sharing, which in this case drives forward another lean principle – perfection of the process [18]. The benefits of SPI include increased customer satisfaction, productivity, quality, cost saving and cycle time reductions.

Statistical control

As Lord Kelvin said: "when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science" [9]. Statistical control means that if the work is repeated in roughly the same way, it will produce approximately the same result. Measurement is the primary principle behind the statistical control. A SDLC model should be under statistical control of project management team. Such a process model will produce the desired results within the anticipated limits of cost, schedule and quality. If the process is not under statistical control, no progress is possible until it is [14].

Suitability to Projects

Not all process models are suitable for every type of projects. Some process models are suitable for large projects, while some may be better for small projects. But, there are some flexible process models which are suitable for different types of projects. Any process model should not be suitable to only a specific type of project or project scope. The model should be designed in such a way that it should support different types of project with their varying scope through process tailoring. Hence, suitability to projects may be considered as an evaluation criteria for a process model.

Support to the Modern Tools and Technologies

Over the time, significant developments are made in the field of new techniques and methodologies. Those are to be incorporated, accommodated and supported in the process models to make it a sustainable for modern software development. If a process model fails to accommodate these new technologies, they gradually become obsolete and useless. For example, during last few years there has been significant development in the field of CASE tools for project management, configuration management, software design, modeling and many more. It is a proven fact that usage of CASE tools increases the product quality and reduces the total project development cost and time to market. Hence, the process model should be designed to support and utilize the CASE tools.

Usability

The usability requirement addresses the various ways in which the SDLC will be used by the team members easily, efficiently while at use. Usability is a composite property of a process model. It is a composition of five attributes i.e. i) Learning ability, ii) Efficiency, iii) User retention over time, iv) Error rate, and v) Satisfaction [19]. Any process model should incorporate these usability attributes. Many usability process has been proposed by several people i.e. ACUDUC (Approach Centered on Usability and Driven by use cases) by Seffah et al [20] , Usability Engineering Process Model (UEPM) by Granollers [21] and Xavier Ferre [19] has proposed an integrated model of usability and different SDLC activities to integrate usability especially in different SDLC models. Adoption Centric Usability Engineering (ACUE) facilitates the adoption of usability engineering methods for software engineering practitioners and thereby improves their integration into existing software development methodologies and practices [22].

5 CONCLUSION

An important initial step in addressing software problems is to treat the entire development process as a performable, controllable, measurable and improved process as a sequence of tasks that will produce the desired result. Any fully effective software process must consider the interrelationships of all the required tasks, tools and methods, skills, training and motivation of the people involved. In general, any process model must bear the properties that are investigated, identified and specified in this paper. At the same time, an ideal SDLC process implementation should be quicker, cost-effective and easy to implement and follow. It should also be stakeholder and project team-friendly. Finally, the ideal

process model should address the top challenges experienced by project managers. Although the degree of importance of these individual characteristics may vary from project to project and depends on situational demand, but we recommend that all the above discussed characteristics should be beared by any process model to suit for the modern real projects.

6 Future Work Plan

In this paper we have identified and enlisted the desired characteristics from any process model irrespective of their degree of importance. In near future, we shall investigate the significance and importance of these individual characteristics and prioratise them accordingly. Further, using such priority list of process model characteristics, we shall develop a process metrics depending on that one may choose the suitable process model for any project which shall provide optimal solution and shall address other common isses related to process models.

REFERENCES

[1] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, MaGrawHill Publications, 2005.

[2] I. Sommerville, *Software Engineering*, Pearson, 2009.

[3] S. L. Pfleeger and J. M. Atlee, *Software Engineering: Theory and Practice*, Pearson, 2011.

[4] A. Mandal, "BRIDGE: A Model for Modern Software Development Process to Cater the Present Software Crisis", *Proc. IEEE Int'l Conf. Advance Computing Conference*, pp. 494-500, 2009. Also available at IEEEXplore with DOI: 10.1109/IADCC.2009.4809259

[5] A. M. Davis, E. H. Bersoff and E. R. Comer, "Strategy of Comparing Alternative Software Development Life Cycle Models", *IEEE Trans. on Software Eng.*, vol-14, no-10, 1988.

[6] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit*, Addison-Wesley, 2003.

[7] Systems Development Life Cycle: Objectives and Requirements, Bender RBT Inc. 17 Cardinale Lane, Queensbury, NY 12804, 518-743-8755

[8] Marcelo Cataldo, James D. Herbsleb; "Coordination Breakdowns and Their Impact on Development Productivity and Software Failures", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 39, NO. 3, MARCH 2013

[9] W. S. Humphrey, Characterizing the Software Process: A Maturity Framework, Technical Report, CMU/ SEI-87-TR-11, ESD-TR-87-112, Software Engineering Institute, 1987.

[10] M. Walton, Strategies for Lean Product Development, Massachusetts Institute of Technology, 1999.

[11] Larman and Vodde, *Scaling Lean and Agile Development: Successful Large, Multisite & Offshore Products with Large-Scale Scrum*, Ch.22, Addison-Wesley, 2008.

[12] KEITH C.C. CHAN , LAWRENCE M.L. CHUNG , Integrating Process and Project Management for Multi-Site Software Development, Annals of Software Engineering 14, 115–143, 2002

[13] M. P. Ginsberg and L. H. Quinn. Process Tailoring and the software Capability Maturity Model. Technical Report, Software Engineering Institute, CMU/ SEI-94-TR-024.

[14] W. Edwards Demming, Quality, Productivity, and Competitive Position, Cambridge, MA: Massachusetts Institute of Technology Center for Advanced Engineering Study, 1982.

[15] Mandal A. and Pal S. C., "Emergence of Component Based Software Engineering", Int'l Journal of Advanced Research in Computer Science and Software Engineeringc, Volume 2, Issue 3, March 2012.

[16] Lowrance, William W., Of Acceptable Risk: Science and the Determination of Safety. Los Altos, Ca: William Kaufmann, 1976.

[17] W. S. Humphrey, "Managing the software process", Reading, Addison-Wesley, MA, 1989.

[18] A. Antanovich, A. Sheyko, B. Katumba, Bottlenecks in the Development Life Cycle of a Feature: A Case Study Conducted at Ericsson AB, Report No. 2010:012, ISSN: 1651-4769, University of Gothenburg.

[19] Xavier Ferré and Natalia Juristo, Helmut Windl, Larry Constantine, "Usability Basics for Software Developers". IEEE Software, January/ February 2001, pg 22-29

[20] A. Seffah, R. Djouab and H. Antunes, "Comparing and Reconciling Usability-Centered and Use Case-Driven Requirements Engineering Processes" 0-7695-0969-W IEEE 2001

[21] Toni Granollers; "User Centered Design Process Model. Integration of Usability Engineering and Software Engineering" Proceedings of INTERACT, 2003

[22] Eduard Metzker, Ahmed Seffah; "Adoption of Usability Engineering Methods: A Measurement-Based Strategy"

Ardhendu Mandal, Assistant Professor, Department of Computer Science and Application, University of North Bengal, India. He has more than 5 Yrs of teaching and research experience in the field of Computer Science and Application with research focus in the field of Software Engineering especially in the area of software development lifecycle process models and its relavent aspects. His research interest also includes High Performance Computing and Bioinformatics.

S. C. Pal, Professor, Department of Computer Science and Application, University of North Bengal, India. He has more than 16 Yrs of teaching and research experience in the field of Mathematics and Compute Science with special research focus in the field of Computational Fracture Mechanics. Recently he is also working in the field of High Performance Computing and Software Engineering.

Volume 2, Issue 3, March 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Emergence of Component Based Software Engineering

Ardhendu Mandal*
Department of Computer Science and Application
University of North Bengal
Pin-734013, India
am.csa.nbu@gmail.com

S. C. Pal
Department of Computer Science and Application
University of North Bengal
Pin-734013, India

Abstract-- **It was noticed that, most software systems are not new but are variants of systems that had been already developed. Hence, a new systems may be developed partially if not completely, from the pre-existing systems by reusing it. This brings the idea of reusability and gave the birth of a noble concept of Component Based Software Development, beyond object oriented development paradigm. Component Based Software Development aims to construct complex software systems by means of integrating reusable software components. This approach promises to alleviate the software crisis at great extents. The objective of this paper is to gain attention towards this new component based software development paradigm and to highlight the benefits and impact of the approach for making it a successful software development approach to the concerned community and industry.**

Keywords— **Software Engineering, Software Components, Component Based Software Engineering, Component Interface**
Abbreviations-- **SE-Software Engineering, CBSE-Component Based Software Engineering, CBSD- Component Based Software Development.**

I. INTRODUCTION

In early days, software engineering approach was ad hoc. Around 1970s, introduction of structured programming" gave a formal shift in software engineering from the ad hoc to a systematic approach. Then around 1980s, introduction of object oriented programming with some advancement explores new areas in software engineering. In recent dates, with the introduction of Component Based Software Development (CBSD), the industry is moving in a new direction. The basic insight is that most software systems are not new. Rather, they are variants of systems that have already been built. This insight can be leveraged to improve the quality and productivity of the software production process [1]. These day's software systems are more complex as compared to those of early. These complex, high quality software systems are built efficiently using component based approach in a shorter time. Component based systems are easier to assemble and therefore less costly to build than developing such systems from scratch. The importance of component based development lies in its efficiency. In addition, CBSE encourages the use of predictable architectural patterns and standard software infrastructure, thereby leading to a higher result. In the remaining part of this paper the term *"component"* and *"software components" will be used interchangeably.*

II. THE JOURNEY OF THE COMPONENT BASED DEVELOPMENT ERA

In 1968, Douglas McIlroy's [2] first share the idea of Component Based Software Development (CBSD) at the NATO conference on software engineering in Garmisch, Germany in his paper titled "Mass Produced Software Components". He discussed the idea that, software may be componentized i.e. built from pre-developed software components. His subsequent inclusion of pipes and filters into the Unix operating system was the first implementation of an infrastructure for this idea. Later, Brad Cox set out to create an infrastructure and market for these components by inventing the Objective-C programming language. IBM led the path with their System Object Model (SOM) in the early 1990s. Some claim that Microsoft paved the way for actual deployment of component software with OLE and COM. As of 2010 many successful software component models do exist.

III. UNDERSTANDING SOFTWARE COMPONENTS

In early days, the principles of software engineering have been focused in developing software system from the very scratch development for individual software. It means that, for all the functionalities to be supported by a system have been designed and coded individually for the proposed systems under development. Brown [3] posits that it would be infeasible for developers and organizations to consider constructing each new information system from scratch.

Instead, information systems would need to be developed with reused practices, software components and products that have been tested and proven to be effective and efficient in order to remain in business and gain competitive advantage [3, 4, 5]. Upon long observation it was found that- there are certain functionalities those are common in many systems. Hence, if these common functionalities can be developed independently - may be reused in different systems without redevelopment from scratch. Later, these can be integrated to any system as part whenever it is suitable! At the same time, it will reduce the development effort of such systems as there is no need to develop the same common parts again and again for different systems. This originates the concepts of Software Components. Although, a software components is a small parts of a system, but often a large system as a whole may be seen as a software component as well. It is also possible that, the system consists of components is a component itself. In all cases, the components are required to be reusable components after all.

Historically, "component" in software is a rough synonym for "module" or "unit" or "routine". A generally accepted view of a software component is that, it is a *software unit with provided services and required services from others too* (Fig. 1). The *provided services* are operations performed by the component whereas, the *required services* are the services needed by the component to provide target services. The one or more *interface* of a component consists of the specifications of its provided and required services [6].

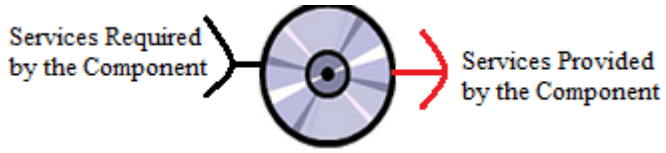


Fig. 1 Service Scheme of Software Component

As component is simply a data capsule, information hiding becomes the core construction principle underlying components. Clemens Szyperski [7] suggests shifting the focus away from code source. He defines a software component as executable, with a black-box interface that allows it to be deployed by those who did not develop it. It is important for a software component to be easily combined and composed with other software components. This is because a software component will only achieve its usefulness when it is used in collaboration with other software components [8].

According to Herzum and Sims [9], the term 'component' is used in many different ways by practitioners in the industry. However, some rather broad and general yet useful definitions are as follows:

"An independently deliverable piece of functionality providing access to its services through interfaces".

-Brown [10]

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

- Clemens Szyperski [11]

A component denotes a self-contained entity (a.k.a. black box) that exports functionality to its environment and may also import functionality from its environment using well-defined and open interfaces. In this context, an interface defines the syntax and semantics of the functionality it comprises (i.e., it defines a contract between the environment and the component). Components may support their integration into the surrounding environment by providing mechanisms such as introspection or configuration functionality.

- Michael Stal [12]

Hence, we may define a software component as: ***An independently deployable and compositional software element having specific functionality that conforms to a software architecture.***

IV. USING SOFTWARE COMPONENTS: COMPONENT BASED SYSTEM DEVELOPMENT

The *usage* of a component in a software system includes using it to replace an out of date component to upgrade the system or a failed component to repair the system, adding it to the system to extend the system services, or composing it into the system while the system itself is still being built. Some researchers insist on a component being reusable during dynamic reconfiguration [13]. CBSD advocates developing software systems by selecting reliable, reusable and robust software components and assembling them within appropriate software architectures.

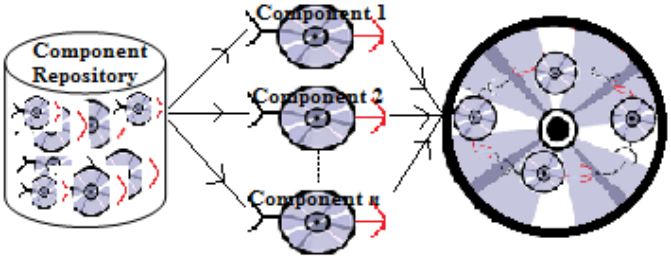


Fig. 2 Component Based System Development

With the help of middleware technologies- a set of specifications or rules in the form of functions, which when incorporated into the code allows the software to be integrated with software developed using other platforms/languages [14]. Example of such middleware technologies are COM, DCOM, CORBA, JavaBeans, EJB etc. Component Based Software Engineering (CBSE) is a process that aims to design and develop software systems using *existing, reusable* and *adaptable* software components as opposed to programming them. Hence, *CBSE shifts the emphasis from programming to composing software systems".*

V. OBJECTIVES OF COMPONENT BASED SOFTWARE DEVELOPMENT: ALLEVIATE SOFTWARE CRISIS

The goal of component-based development is to build and maintain software systems by using existing software components [15, 16, 17, 18, 19, 20]. As said by Dr. Randall W. Jensen [21], "High customer demand, reduced software development budgets, and a competitive software market drive the need for reusable software". When correctly applied and implemented, developing software systems using Commercial Off The Shelf (COTS) software components promises benefits like increase productivity, shorten time-to-market, improve software quality, reduce maintenance cost, allow for inter-application interoperability, decreased level of risks, leverage technical skills and knowledge, and improve system functionality [22, 23, 24]. As, software crisis may be loosely defined as the set problems associated with the software development process [25] i.e. quality, development cost and time-to-market of software, we may conclude that the indirect objective of CBSD are to alleviate software crisis. In addition, the particular objectives of software components are to:

a. **To have a useful 'replaceable property'** i.e. easy to assemble and easy to disassemble.

b. **Increase Reusability:** Develop once and reuse several instances of the same over the period.

c. **Facilitating System Change Management and System Maintenance: T**he plug and play feature of a component allows easy component composition and inclusion in the information systems.

d. **Enhancing Development Flexibility: C**omponents are an independent software element that can be designed and developed independently enhancing the development flexibility.

e. **Reduced System Development Time: R**eusing pre-developed existing components instead of new fresh development will reduce total development time.

f. **Reduced System Development Cost: R**educed development time will result in significant reduction in total development cost.

g. **Improve Software Quality:** Ideally, a component is pre-tested for errors and quality parameters. Hence, using such pre-tested, high quality software components improves the quality of complete software systems.

h. **Reducing Project Risk:** From management perspective, if an asset's costs can be optimized through a large number of uses, it would then be possible for the management to expend more effort and allocate more budgets to improve the quality of software components. This in turn reduces the level of risk faced by the development effort and will undeniably improve the likelihood of success [26].

i. **Improve interoperability: W**hen systems are developed using reused components, they are expected to be more **interoperable** as they rely on common mechanisms to implement most of their functions [27]

j. **Increase System Learning for User:** Dialogs and interfaces used by these systems would be similar and would improve the learning curve of users who utilize several different systems built using the same components [26, 27].

k. **Ease and Efficient System Debugging:** As the components are previously tested, if any error occurs, must be during the integration. Hence, the domain and range for debugging is minimized and localized. This facilitates the debugging process quite easy and efficient.

VI. IMPACT COMPONENT BASED SOFTWARE DEVELOPMENT: AN QUANTITATIVE ANALYSIS

The CBSD approach includes improvements in: quality, throughput, performance, reliability and interoperability; it also reduces development, documentation, maintenance and staff training time and cost [23]. In this approach, due to inherent functional independence software is assembled from components can be autonomously deployed and, the productivity and performance of the development team can be improved [9]. The impact of software reuse in system development is highlighted below:

a) Impact on Productivity

Although percentage productivity improvement reports are notoriously difficult to interpret, it appears that 30-50% reuse can result in productivity improvements in the 25–40% range. According to Lim [22], Hewlett-Packard software projects reported productivity increases from 6% to 40% with the incorporation of CBSD. Further, Pitney Bowes in the USA which has been reusing components since 1996 documented tremendous savings in labour as the company is now able to achieve 500 human-weeks of development progress in only 200 human-weeks by using existing components and by purchasing others from component markets [28].

b) Impact on Quality

In a study conducted at Hewlett Packard, Lim [22] reports that the defect rate for reused code is 0.9 defects per KLOC, while the rate for newly developed software is 4.1 defects per KLOC. Henry and Faller [27] reported that, for an application that was composed of 68% reused code, the defect rate was 2.0 defects per KLOC—a 51% improvement from the expected rate, had the application been developed without reuse. They further report a 35% improvement in quality by component reuse in system development. They again suggested that, the quality of information systems developed using this approach will also have fewer bugs and defects if compared with newly built-from-scratch systems.

c) Impact on Time-to-Market

The **STG** division reports that the same development effort using the reusable work product required only 21 calendar months compared to an estimated 36 calendar months had the reusable work product not been used, a reduction of 42% [22].

d) Impact on Cost

Apart from productivity gains, component reuse allows organizations to reduce the critical path in the delivery of

software systems, reducing the time-to-market and begin to accrue profits earlier. Study shows that there has been reduction in product cost up to 75-84% as a result of reuse [29].

VII. INDUSTRIAL PRACTICES ON SOFTWARE COMPONENTS

"The use of commercial off-the-shelf (COTS) products as elements of larger systems is becoming increasingly commonplace. Shrinking budgets, accelerating rates of COTS enhancement, and expanding system requirements are all driving this process. The shift from custom development to COTS-based systems is occurring in both new development and maintenance activities. If done properly, this shift can help establish a sustainable modernization practice."

- SEI COTS-Based Systems Initiative [30]

Because the potential impact of reuse and CBSE on the software industry is enormous, a number of major companies and industry consortia have proposed standards for component software. In recent years, component technologies have been well developed, such as Enterprise Java Beans (EJB) of Sun [31], CORBA Component Model (CCM) of the OMG [32], and Component Object Model (COM) of Microsoft [33].

A. *SUN JavaBeans Components*

The JavaBean [33] component system is a portable, platform independent CBSE infrastructure developed using the Java programming language. The JavaBean system extends the Java applets to accommodate the more sophisticated software components required for component-based development. The *Bean Development Kit* (BDK) encompasses a set of tools to facilitate the CBSD approach.

B. *OMG/CORBA*

The Object Management Group has published common *object request broker architecture* (OMG/CORBA) [32]. An object request broker (ORB) provides a variety of services that enable reusable components to communicate with other components, regardless of their location within a system. When components are built using the OMG/CORBA standard, integration of those components without modification in a system is assured if an *Interface Definition Language* (IDL) is created for every component.

C. *Microsoft COM*

Microsoft has developed a Component Object Model (COM) [35] that provides a specification for using components produced by various vendors within a single application running under the Windows operating system. COM encompasses two elements: *COM interfaces* that are implemented as COM objects and a set of *mechanisms for registering and passing messages* between COM interfaces. From application point of view, "the focus is not on how implemented, only on the fact that the object has an interface that it registers with the system, and that it uses the component system to communicate with other COM objects [34]".

VIII. CONCLUSIONS

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. Developing software using Commercial Off The Shelf reusable component is known as Component Based Software Development (CBSD). Informally, application of Software Engineering principles and practices in CBSD is known as Component Based Software Engineering (CBSE). CBSD qualifies, adapts, and integrates software components for reuse in a new system. In addition, often engineers need to develop additional components that are based on the custom requirements of a new system that are unavailable from component library. CBSD offers inherent benefits in software quality, developer productivity, and overall system cost, but yet many roadblocks remain to be overcome before the CBSD is widely used throughout the industry. We may conclude that, component based development is the future development process to cater the present software crisis. Industries must follow this development practices, built and enlarge their component library for future reuse. At the same time industries must train their developers to encourage and practice the component based software development process and need to set up appropriate facility centres for supporting CBSD process. Despite lots of potentialities, there are still lots of issues that are to be explored for being the CBSD successful. Hence, researchers have to take the responsibilities on their shoulder to resolve these issues and make CBSD a successful software development approach.

REFERENCES

[1] William B. Frakes and Kyo Kang, Software Reuse Research: Status and Future, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 31, NO. 7, JULY 2005

[2] M. D. McIlroy, Mass Produced Software Components , NATO Software Engineering Conference Report, Garmisch, Germany, October, 1968, pp. 79-85.

[3] Brown, A. W., 'Preface: Foundations for component-based software engineering', in *Component-based Software Engineering - Selected Papers from the Software Engineering Institute*, A. W. Brown (ed), 1996, pp. vii - x.

[4] Butt, J., 'Reuse Rather Than Rebuild', *eWeek*, 18(44), 2001, p. 37.

[5] Szyperski, C., *Component Software - Beyond Object-oriented Programming*, ACM Press/Addison Wesley, USA, 1998.

[6] Kung-Kiu Lau and Zheng Wang, *Software Component Models*, IEEE Transactions on Software Engineering, Vol. 33, NO. 10, October 2007.

[7] Clemens Szyperski, *Component Software*, Addison-Wesley, 2nd edition, 2002.

[8] Councill. B. & Heinernan, G. T., 'Definition of a Software Component and its Elements', in Component-based Software Engineering - Putting the Pieces Together, B Councill & G. T. Heineman (eds), Addison Wesley, USA, 2001.

[9] Herzum, P. & Sims, O., Business Component Factory- A Comprehensive Overview of Component-based Development for Enterprise, John Wiley, New York, 2000.

[10] Brown, A. W., Large-Scale, *Component-Based Development*, Prentice-Hall, USA, 2000.

[11] *WCOP'96 Summary in ECOOP'96 Workshop Reader*, Dpunkt Verlag, 1997. ISBN 3-920993-67-5.
[12] Anton Deimel, Juergen Henn, et al., *What characterizes a (software) component? Software - Concepts & Tools*, Vol. 19, No. 1. (1 June 1998), pp. 49-56.
[13] M.R.V. Chaudron and E. de Jong. *Components are from Mars.* In Proc. 15 IPDPS 2000 Workshops on Parallel and Distributed Processing, Lecture Notes In Computer Science; Vol. 1800, pages 727-733, 2000.
[14] Sparling M.: "Is there a Component Market", www.cbd_hq.com/articles/2000
[15] G. Gossler and J. Sifakis. *Composition for component-based modeling*, Science of Computer Programming, 55(1-3), 2005.
[16] N. Medvidovic and R.N. Taylor. *A classification and comparison framework for software architecture description languages.* IEEE Transactions on Software Engineering, 26(1):70.93, 2000.
[17] R. Roshandel, B. Schmerl, N. Medvidovic, D. Garlan, and D. Zhang. *Understanding tradeoffs among different architectural modeling approaches.* In Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA04). IEEE Computer Society, 2004.
[18] J.-G. Schneider and O. Nierstrasz, *Components, scripts and glue.* In L. Barroca, J. Hall, and P. Hall, editors, Software Architectures Advances and Applications, pages 13-25. Springer, 1999.
[19] D. Hybertson. *A uniform component modeling space*. Informatica, 25:475.482, 2001.
[20] Szyperski, C., *Component Software - Beyond Object-oriented Programming*, ACM Press/Addison Wesley, USA, 1998.
[21] Dr. Randall W. Jensen, An Economic Analysis of Software Reuse, CROSSTALK The Journal of Defense Software Engineering December 2004
[22] Lim W.C., *Effect of reuse on quality, productivity and economics*, IEEE Software Vol 11, No 5, Sep 1994, pp23-30
[23] Pinto, P. E., *Promoting Software Reuse in a Computer Setting* [Online], Available: http://www-2.cs.cmu.edu/afs/cs/usr/ppinto/www/ reuse.html, [Accessed: 20 August 2002].
[24] Patrizio, A., 'The New Developer Portals - Buying, selling, and building components on the web speeds companies' time to market', Information Week, August, p. 81, 2000.
[25] Ardhendu Mandal, *BRIDGE: A Model for Modern Software Development Process to Cater the Present Software Crisis*, 2009 IEEE International Advance Computing Conference (IACC 2009) Patiala, India, 6–7March 2009
[26] Lim, W. C., *What is Software Reuse?* [Online], Available: http://www.flashline.com/content/lim/what_reuse.jsp, [Accessed: 20 August 2002].
[27] Henry E., Faller B., *Large-scale industrial reuse to reduce cost and cycle time*, IEEE Software, Vol 12, No 5, Sep 1995, pp47-53
[28] Scannell, E., *Web Services Reignite Component Reuse* [Online], Available: http://www.itworld.com/AppDev/4162/IWD010409hnreuse/pfindex.html, [Accessed: 15 August 2002]
[29] Kuljit Kaur, Parminder Kaur, Jaspreet Bedi, and Hardeep Singh, Towards a Suitable and Systematic Approach for Component Based Software Development, World Academy of Science, Engineering and Technology 27 2007
[30] Robert C. Seacord, Daniel Plakosh, Grace A. Lewis, Modernizing legacy systems: Software Technologies, Engineering Processes and Business Practices
[31] Sun Microsystems. Enterprise JavaBeans(TM) specification 2.0. *Sun Developer Network Enterprise JavaBeans Technology Official Website*. Sun Microsystems, Inc.: Santa Clara CA, 2005; 572 pp. Http://Java.sun.com/products/ejb [10 September 2004]
[32] Object Management Group. CORBA component model (CCM) 3.0. *Object Management Group Working Group Specification*. Object Management Group, Inc.: Needham MA 2002; 434 pp. Http://www.omg.org/cgibin/apps/doc?formal/02-06-65.pdf [10 September 2004]
[33] Rogerson D. *Inside COM*. Microsoft Press: Redmond WA, 1997; 376pp.
[34] Carlos Canal and Antonio Cansado, *Component Reconfiguration in Presence of Mismatch*, Informatica 35 (2011), Pages 29–37

Proceedings of the 4th National Conference; INDIACom-2010
Computing For Nation Development, February 25 – 26, 2010
Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi

SRS BUILDER 1.0: An Upper Type CASE Tool For Requirement Specification

Ardhendu Mandal
Department of Computer Science and Applications, University of North Bengal (N.B.U.), INDIA
E-Mail:am.csa.nbu@gmail.com

ABSTRACT
Software (SW) development is a labor intensive activity. Modern software projects generally have to deal with producing and managing large and complex software products. Developing such software has become an extremely challenging job not only because of inherent complexity, but also mainly for economic constraints unlike time, quality, maintainability concerns. Hence, developing modern software within the budget still remains as one of the main software crisis. The most significant way to reduce the software development cost is to use the Computer-Aided Software Engineering (CASE) tools over the entire Software Development Life Cycle (SDLC) process as substitute to expensive human labor cost. We think that automation of software development methods is a valuable support for the software engineers in coping with this complexity and for improving quality too. This paper demonstrates the newly developed CASE tools name "SRS Builder 1.0" for software requirement specification developed at our university laboratory, University of North Bengal, India. This paper discusses our new developed product with its functionalities and usages. We believe the tool has the potential to play an important role in the software development process.

KEYWORDS: CASE tool, software development, SDLC, SRS, SE, FHD.

1. INTRODUCTION

Although, hardware costs are decreasing drastically, still the computers are not used extensively by the business organization because of the huge software cost. In recent years, Computer-Aided Software Engineering tools have emerged as one of the most important innovations in software development to manage the complexity of software development projects reducing its product cost. Using CASE tools over their SDLC process may reduce the developing cost significantly.

Although, almost all develop countries do use certain CASE tools, but its extensive use is still a dream because of their product cost. Although, big software giants do use their own developed or commercially available CASE tools in development software, but their quality, applicability, cost, availability remains as big question. The huge cost of such commercially available CASE tools made these outreach of the small software development companies. In this study, we are going to demonstrate newly developed requirement specification tool name SRS BUILDER version 1.0.

The rest of the paper is organized as follows: We started by defining software engineering, Computer Aided Software Engineering and CASE tools. Then, we have laid down the different types of CASE tools and advantage of using CASE tools with its limitation. In the later section we have discussed about SRS BUILDER 1.0 with its sample design.

2. COMPUTER AIDED SOFTWARE ENGINEERING (CSAE) and CASE TOOLS

In the following we are going to define some terminologies used in software engineering.

- **Software Engineering (SE):** Before defining CASE, defining software engineering is justifiable. Although, the age of *Software Engineering* is quite old, but prior to its born, people do used to develop software. But because of some problems (discussion of these problems are beyond the scope of paper) faced later with those systems, software engineering emerged as a new subject.

The IEEE defines ***software engineering*** as [6]:

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

More significantly, software engineering may be defined as the systematic approach to develop **quality** software **within both budget and time** constraints.

- **Computer-Aided Software Engineering (CASE):** CASE is an acronym that stands for Computer-Aided Software Engineering. Roughly, this is all about using computers at different phases of the SDLC process during the development and maintenance of software to assistance the development team. CASE provides the software engineer with the ability to automate manual activities and to improve engineering associated to software development. Basically, *it is all about using software to develop software.*
- **CASE tools:** CASE tools are the tools that permit collaborative software development and maintenance. These tools are concerned with automated tools that aid in the definition and implementation of software systems.

Formal definition of CASE:

- "Individual tools to aid the software developer or project manager during one or more phases of software development (or maintenance)."
- "A combination of software tools and structured development methodologies".

- **Types of CASE tools:** Depending on the activities performed, CASE tools are primarily divided in to three categories. They are as follows:
  - **Upper CASE tools:** Primarily focuses on the System

 ISSN 0973-7529 ISBN 978-81-904526-9-4

analysis and design phase of the SDLC.

- **Lower CASE tools:** Focuses on system implementation phase of SDLC.
- **Integrated CASE tools:** It helps in providing linkages between the lower and Upper CASE tools.

3. ADVANTAGES OF USING CASE TOOLS

With the growing importance of CASE tools, more steps in the SDLC are being automated. However, a complete automated software facility for all steps is still a dream. Presently available CASE tools cover only a certain modules of the general CASE facility.

The benefits of using CASE tools are as follows:

- Increasing Productivity.
- Product Quality improvement.
- Development Cost Reduction.
- Effort Reduction: Different studies carried out to measure the impact of CASE put the effort reduction about 30-40% [4].
- Reduction in Development Time.
- Reduce the drudgery and working style in a software engineer's work.
- *Create good quality documentation. Our, proposed tool basically focuses on this particular aspect of computer aided software engineering.*
- Create maintainable system.
- Providing a uniform platform for software/system developers to present information and knowledge compactly for ease of communication (Banker & Kauffman, 1991; Church & Matthews, 1995; Orlikowski, 1989).

4. USAGE OF CASE TOOLS: A RESEARCH REPORT

CASE (Computer-Aided Software Engineering) tools are supposed to increase productivity, improve the software

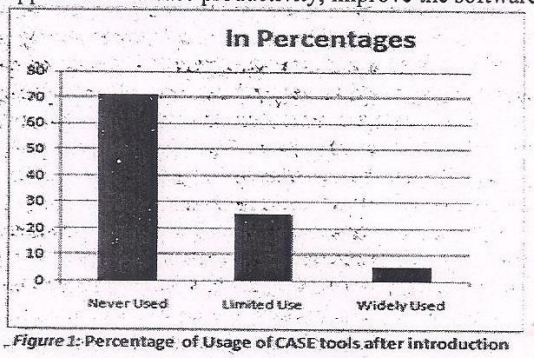


Figure 1: Percentage of Usage of CASE tools after introduction

product quality and make Information Systems development a more enjoyable task [2]. However, they have been failing to deliver the benefits they promise [1]. The results of the research carried out by Kemerer, C. F. [3] regarding the usage of CASE tools after introduction are shown in the *Figure 1.*

This results about CASE tool usage raise a significant question, "Why the percentage of widely used case tools are too low (5% only)?" The answer to the question is the limitations with the CASE tools discussed in the next section.

5. LIMITATION OF CASE TOOLS

CASE tools are still in limited use because of the following limitations:

- CASE tools don't support automatic development of functionally relevant system.
- It force system analyst to follow a prescribed methodology.
- It may change the system analysis and design process.
- Poorly supported expected functionality from them.
- Huge cost of the CASE tools.
- Lack of Concern in CASE tool usage.
- Bad quality of the CASE tools.
- CASE tools are complex because they offer a large array of options and support for software development activities.
- Cheap Labour cost: In developing countries like India and other underdeveloped countries, cheaply available human resources remain as one of the biggest reasons for not using the costly CASE tools.

6. MOTIVATION FOR UNDERTAKING THE RESEARCH PROJECT

In addition to usage in industry by practitioners, CASE tools are employed by educators to teach students software development skills. Evaluating and selecting a CASE tool for a specific systems development course is quite difficult.

While teaching the paper software engineering at university (NBU), it was found that the students are very much confused about CASE tools. The significant reasons for the same are basically widely available complex poor quality CASE tools which are quite expensive to purchase although. Some modules are good in some if not in all. The students do not feel interesting to use them. Then we under took the project to develop a new customised CASE tool for requirement specification supporting IEEE specification as per the students need that will help the student to improve their skill and have a clear vision about it. We start with the development of CASE tool to specify the system requirements to generate the System Requirement Specification (SRS) document. The outcome of our attempt is the so named, "***SRS BUILDER 1.0***"- the CASE tools to generate the SRS document. Moreover, we are planning to distribute the newly developed CASE tool to the educational institutions on demand almost free of cost with a nominal transportation cost for the sake of student community.

7. SRS BUILDER 1.0: THE NEW REQUIREMENT SPECIFICATION TOOL

As from the SLC models, we know that after the requirements are gathered by the system analyst, the analysed and finalised requirements need to be specify in the SRS document. The SRS document is then provided to the software development team for next phase of the development.

- ***SDLC Model Followed:*** We have followed the **BRIDGE** software development process model [5] to develop the *SRS BUILDER 1.0.*

 ISSN 0973-7529 ISBN 978-81-904526-9-4

- ***Product Quality Features:***
  - Support to IEEE specification for SRS writing format along with the flexible other customized specifications as per your organizational need.
  - Good Graphical User Interface
  - Easy to use
  - Easy Installation
  - Incorporated User Documentation
  - Easily Available
  - Compatible
  - Minimal System Requirements for Installation
  - Validated
- ***System Specification:***
  Front End: Visual Basic 6.0
  Back End: MySQL
  Operating System: Windows XP, Windows Vista.
  Memory Requirement: 15 KB

8. FUNCTION HIERARCHY DIAGRAM (FHD) OF SRS BUILDER 1.0

The Function Hierarchy Diagram (FHD) of the SRS BUILDER 1.0 is shown below in *Figure 2.*

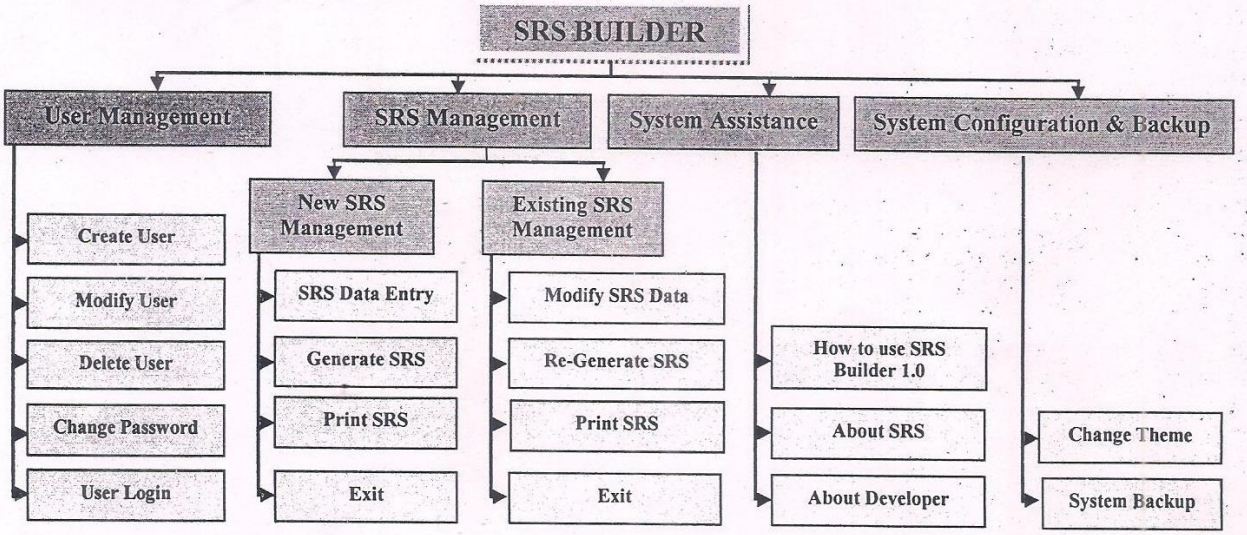


Figure 2: *Function Hierarchy Diagram (FHD) of SRS Builder 1.0*

9. SAMPLE SRS ORGANIZATION GENERATED BY SRS BUILDER 1.0

A typical SRS generated by the SRS BUILDER 1.0 for the ATM system is shown below in *Figure 3*. This is not a complete and correct SRS for the intended system, but a sample used only to show the SRS organization generated by the tool. The font size and spacing has been changed to accommodate in the paper keeping the context organization unchanged.

Project Title: SRS ATM

Project Id: 1

SOFTWARE REQUIREMENT SPECIFICATION

Introduction

Purpose:

This document describes the software require……

Scope:

The software supports a computerized banking …….

Definition:

 ISSN 0973-7529 ISBN 978-81-904526-9-4

• Account:

A single account at a bank against which transaction……..

Intended Audience:

The intended audience of this SRS consists of:

• Software designers….

Reference: NA

Overview: NA

Document Conventions: NA

Overall Description

Product Perspective:

An automated teller machine (ATM) is a……. customer is identified by inserting a plastic ATM card ……..

Product Function:

1. Get Balance Information…..

User Characteristics:

Open to all authorized users..... Customers are simply members of the public with no special training……

Operating Environment: Ability to read the ATM card……

General Constarints: NA

User Documentation: NA

Assumptions Dependencies: Hardware never fails ……

Specific Requirements

N.A……

External Interface Requirements

User Interface:

The customer user interface should be intuitive, such that 99.9% of all new ATM users are able……..

Hardware Interface:

• Ability to read the ATM card.....

Software Interface:

• State Bank……

Communication Interface:

• List of Communicational interface equirements ……

Functional Requirements:

• List of functional requirements ……

Behavioural Requirements:

 ISSN 0973-7529 ISBN 978-81-904526-9-4

• List of behavioural requirements .....

Other Non-functional Requirements

Performance Requirements:

• It must be able to perform in adverse conditions like high/low temperature etc. .....

Safety Requirements:

• Must be safe kept in physical aspects, say in a cabin .....

Security Requirements:

• Users accessibility is censured in all the ways .....

Software Quality: NA

Other Requirements: NA

SYSTEM REQUIREMENTS SPECIFICATION for ATM Withdrawal

Submitted by:

Program Manager/Functional Project Officer	Date
Coordination:	
Director, Applications Architecture	Date
Director, Engineering	Date
Test Director	Date
Approved by:	
Functional Manager	Date

Figure 3: A typical SRS organization generated by SRS BUILDER 1.0

10. CONCLUSION

We may conclude the paper by pointing out that, this CASE tool will play an important role to the software developers and learners to use and understand the utility of the CASE tool in today's complex software projects. Also, as we mentioned earlier, interested educational institutions and organizations may contact the author for the CASE tool for their usage.

11. FUTURE WORK

SRS BUILDER 1.0 CASE tool has been tested and validated properly. In future, we propose to enhance the capabilities of the present version by appending the functionalities to design the various UML diagrams.

12. REFERENCES

[1]. Ilvari, J. (1996). '*Why are CASE Tools Not Used?*". Communications of the ACM, 39:94-103.

[2]. Jarzabek, S. and Huang, R. (1998). '*The case for User-Centered CASE tools*", Communications of the ACM, 41(8): 93-99.

[3]. Kemerer, C. F. (1992). "*How the Learning Curve Affects CASE Tool Adoption*". IEEE Software, 9, 23-28.

[4]. Mall, Rajib. "*Fundamentals of Software Engineering*", Second Edition, PHI.

[5]. Mandal, Ardhendu. "*BRIDGE: A Model for Modern Software Development Process to Cater the Present Software Crisis*", Proceeding, PP: 494-500, IEEE International Advance Computing Conference (IACC 2009), Patiala, India, 6–7March 2009, ISBN-978-981-08-2465-5

[6]. Roger S. Pressman, "*Software Engineering: A Practitioner's Approach*", Mc-Grawhil, Sixth Edition, 2005.

 ISSN 0973-7529 ISBN 978-81-904526-9-4

2009 IEEE International Advance Computing Conference (IACC 2009)
Patiala, India, 6–7 March 2009

BRIDGE: A Model for Modern Software Development Process to Cater the Present Software Crisis

Ardhendu Mandal
Lecturer, Department of Computer Science and Application, University of North Bengal
Raja Rammohanpur, PO-NBU, Dist-Darjeeling, West Bengal, Pin-734013, India.
am.csa.nbu@gmail.com

Abstract-*As hardware components are becoming cheaper and powerful day by day, the expected services from modern software are increasing like any thing. Developing such software has become extremely challenging. Not only the complexity, but also the developing of such software within the time constraints and budget has become the real challenge. Quality concern and maintainability are added flavour to the challenge. On stream, the requirements of the clients are changing so frequently that it has become extremely tough to manage these changes. More often, the clients are unhappy with the end product. Large, complex software projects are notoriously late to market, often exhibit quality problems, and don't always deliver on promised functionality. None of the existing models are helpful to cater the modern software crisis. Hence, a better modern software development process model to handle with the present software crisis is badly needed. This paper suggests a new software development process model, BRIDGE, to tackle present software crisis.*

I. Introduction

Now a day, computers running with special purpose application software are being used as an extensive aid to solve complex problems almost each and every place starting from gaming to engineering, industries applications, scientific research and different allied fields. These special purpose softwares are some times unique and distributed in nature with higher degree of complexity. Developing such complex software is not so easy because of the different constraints. Our existing software models do not provide adequate flexibility to be applied for such large and complex projects. So we must have a better software development process model that will help to overcome these challenges.

II. Usage of Different Process Models: A Survey Report

The result of the survey carried out by Dr. Jon Holt [3], related to current practice in software engineering reveals the percentage of usage of different types of software development lifecycle models (SDLC) in practice. The result is shown below in *Figure 1*. Although different organizations do use different lifecycle models, but from the above data it is clear that *a large part of industries (22%) do not use any lifecycle model at all!* The BIG question here is why these organizations do not follow any life cycle model?

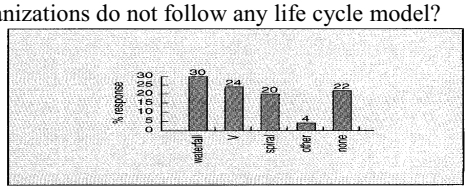


Figure 1: Use of different SDLC models in Practice [3]

The probable answer is, either no lifecycle model is suitable for their projects or they don't find it useful. In either case, it means the existing models lacking suitability. Hence, we need to improve the suitability of these models so that it can be used in practice.

III. Characteristics of good Software Development Process Model

Any software development process model should have the following characteristics [2] for quality software development:

i. *The project goal reflection* i.e. the process model must reflect the project development goals.

ii. *Predictability i.e.* it must be able to forecast the out put of the project following the model prior to project completion.

iii. *Support testability and maintainability* i.e. the process model must focus on reducing the cost, effort of testing and maintenance.

iv. *Support change* i.e. the process model must handle the necessary changes.

v. *Early Defect Removal,* because the delay in error detection increases the costs to correct them.

vi. *Process improvement and feedback* i.e. each project done using the existing process model must feed information back to facilitate further process improvement.

vii. *Quantitative progress measurements* i.e. the process model at any point must give a quantitative measurement of the progress attained.

viii. *Support of process tailoring* in special situations at necessity.

IV. Nature of Modern Software Projects

The earlier software projects were of limited scope with relatively less complexity and smaller size. In contrast, the modern software has *wider scope, higher degree of complexity and larger size with better quality, portability* and *scalability* requirements. Some times, the modern software has *to work with some existing legacy system*. Developing such system are more challenging because of the *inter-operatability* and *dependency* factors. The modern real-time systems have lots of *critical issues* such as *time and space complexity* requires to be addressed. Tremendous hardware development rate has brought us towards the system-on-chip (SOC) era. In such systems, the software has to work in coordination with the particular hardware. Developing such systems are more critical because of the hardware *constraints*. As result of advancement in network technology, more often systems are becoming *web based* and *distributed* in nature. In conclusion, the modern softwares are different in various respects from the earlier softwares.

V. Modern Software Crisis:

Software Crisis may be loosely defined as the *problems associated with the software development process*. Among a lot, a few critical software crisis with modern software development are listed below [5,8]:

i. Larger *size.*
ii. Increasing *complexity.*
iii. Higher development *cost*.
iv. The *delivery* challenges i.e. *late* system delivery.
v. The *trust* challenge. How much can we trust on system operations?
vi. *Incorrectness*: Not satisfying the client needs exactly.
vii. Poor *quality.*
viii. Poor *productivity.*
ix. The *heterogeneity* Challenges i.e. inter-system coordination problem.
x. Demand of *reusability.*
xi. *Modularity*.
xii. *Maintainability*.
xiii. *Integration* problem.
xiv. *Scalability*.
xv. *Portability*.
xvi. *Change* Management.
xvii. *Risks* associated with software development.

VI. Trends in Modern Software Development

Recently, lots of new approaches are being used at practice to overcome the modern software crisis. Some recent trends in modern software developments are listed below:

i. Component based software development.
ii. Software reuse
iii. Aspect oriented software development.
iv. Service oriented software development.
v. Multi-Tired Software Design.
vi. Object Oriented Software Development.
vii. Standards practices.
viii. Use of CASE tools.

VII. Reasons for failure of Traditional SDLC Models: The Shortcomings

After analysing the existing SDLC models, the shortcoming of these models may be broadly summarised as follows:

i. *Non-Involvement of the client* over the entire project development.
ii. *Lack of better understanding* of the system requirements.
iii. *Lack of communications* among the team members.
iv. *Lack of project management* controls over the entire development period.
v. *Overlooking verification* activity
vi. *Insufficient documentations*.
vii. *Lack of configuration management*.
viii. *Non importance to component based* software *development and*
ix. *Poor support of component reusability*.

Directly or indirectly, the above reasons are the real causes of the various software crises. I have tried to address these causes of software crisis in my proposed model discussed shortly.

VIII. Need of Modified Process Model

Although, tailored traditional software development process models are being used since a long time, but these are not good enough at practice. Hence, we are in search of a new software development process model that will adopt and encourage these modern practices. In the forth-coming section a rather novel software development process model-BRIDGE, is proposed and discussed that attempts to encourage the modern software development trends. As well said by David Norton, research director at Gartner "I do not feel waterfall development was bad. It's given us a lot of software over the last 30 years, but I think its time is up"[1].

IX. BRIDGE: The Proposed Model for Modern Software Development Process

After analysing the importance of all the recent software development trends, at attempt is taken to develop a rather new and novel software development process model that adopts the modern software development trends and practices. The so named BRIDGE model is the result of such an attempt, which is elaborated over the following sections. The schematic diagram of the BRIDGE model is given in *Figure 2*.

A. BRIDGE Process Model Description:

Unlike the other process models, the BRIDGE model consists of several phases with distinguished objectives that are discussed in the following section briefly:

i. Phase1: Requirement Analysis, Verification and Specification

The *objective* of this phase is to *identify the exact requirements* from the client using different techniques and to specify them in a document for future use after verification. During *requirement gathering,* the analyst extracts the system requirements from the client. In practice, it is really a tough job for the analyst to extract the requirements from the client, as the clients are unable to identify and express the exact requirements prior experiencing the system practically. The gathered requirements required to be analyzed for removing the redundancy, incompleteness, inconsistencies, anomalies etc. This phase is often called the *requirement analysis phase.* Finally, the verified requirements are to be *specified* in a document called *Software Requirements Specification (SRS)* and stored for future use. This phase is often called *requirement specification* phase. This SRS document may serve as the agreement document between the client and the company and becomes the baseline for proceeding to the next phase.

ii. Phase2: Feasibility Analysis, Risk Analysis, Verification and Specification

The *objectiv*e of this phase is to *analyze the suitability* of the project in respect to different project attributes to check the different suitability aspects among the alternatives. After carrying out the analysis, the optimal solution is selected. At this stage the project cost estimation has to carry out. The different feasibility i.e. *economic feasibility*, *technical feasibility*, *operational feasibility* has to carry out to manage the different system constraints. Some times, the result of the different feasibility analysis may contradict. In such cases, necessary changes, modification and/or negotiation may have to do in the project upon consulting the client if the project is not cancelled. Finally, after verification the result of the feasibility analysis has to be specified in a document called *feasibility report and to be kept for future reference.* Beside feasibility analysis, at this phase the different project risks have *to be identified, analyzed* and specified in the *risk specification document.*

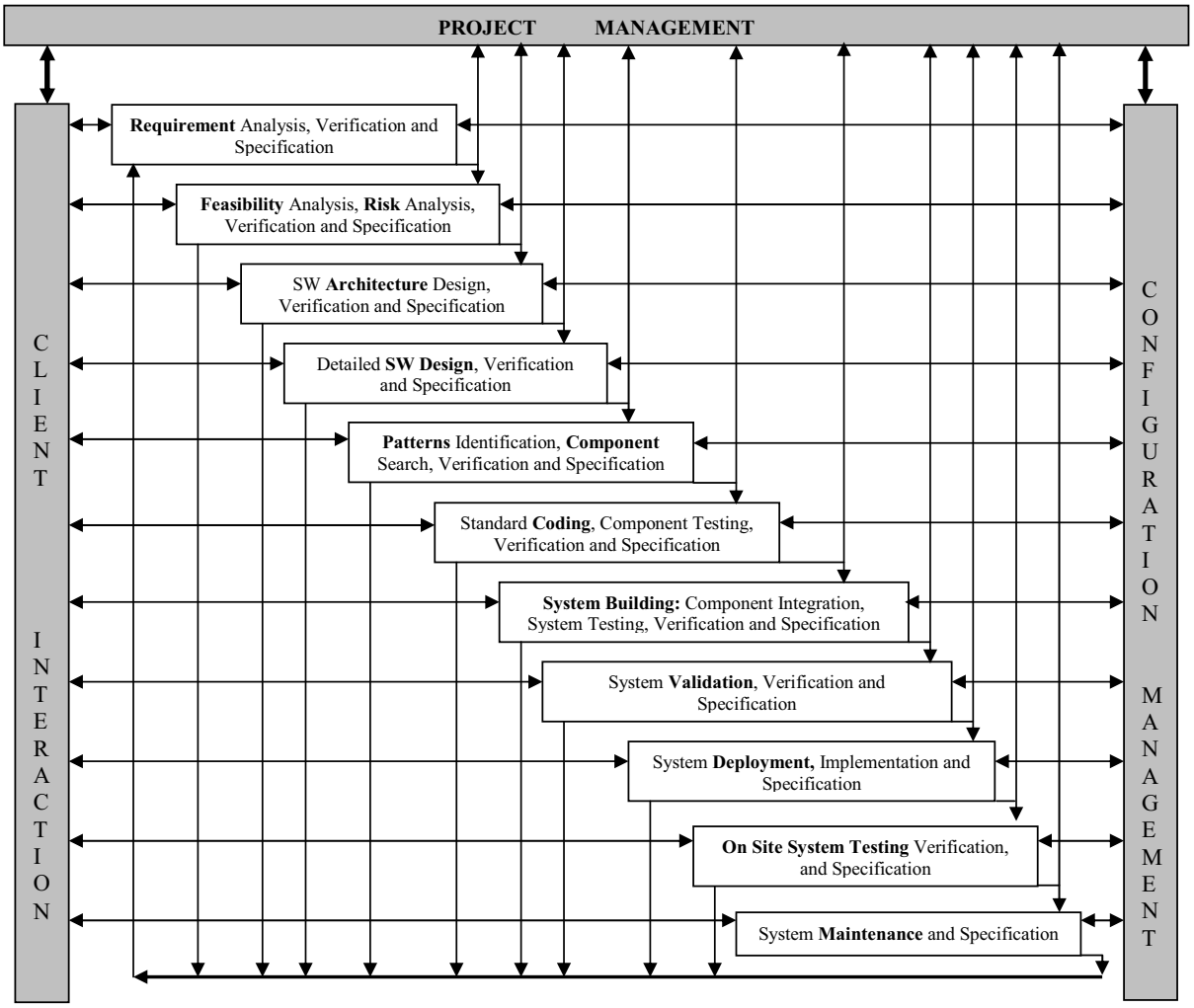


Figure 2: BRIDGE Software Development Process Model

iii. Phase 3: Software Architecture Design, Verification and Specification

Once the project is confirmed, we must design the software architecture. Software architecture design is a *high-level design activity* and relatively a recent trend in industries after understanding its importance. We may consider software architecture as *abstract design* of the complete system. The *objective* of software architecture design is to *identify the sub-systems, building blocks or the components* of the system along with their *communication interfaces* expressing their *external behavior* to improve the project understandability and to communicate with the different stakeholders. The architecture design should reflect the functional requirements specified in the SRS document. Once the software architecture is designed, the architecture design must be *verified* to check whether it conforms the system requirements correctly or not. The verified software architecture design is specified in a *software architecture design document (SADD)*. It must be clear that implementational issues are not considered while designing the software architecture.

iv. Phase 4: Detailed Software Design, Verification and Specification

In this phase, the *detailed design* of the system has to be prepared conforming the software architecture designed during the last phase. Software design is basically a *low-level design* activity *keeping the implementational issues in mind*. The *objective* of this phase is to prepare the *modular design* of the system that can be *directly implemented using some programming language*. The *data structure and algorithms* are also to be developed in this phase. The verified software design specified in a document named as *software design document (SDD)* that will be used in the other development activities later.

v. Phase 5: Patterns Identification, Component Search, Verification and Specification

In general, a system consists of a set of sub-systems, so called *components*. If we analyze any problem, we may find some components common in different projects *representing some general structures of a system.* These common components are sometimes called *patterns.* The *objective* of this phase is to *identify these patterns*. But, to use these pre-developed components efficiently in our system, the *system must be designed keeping this objective* in mind and the designer should be well aware of the available components in the component library. From the architecture design, we must

be able *to identify the components* and then it must be *searched* in the component library to find a suitable *component match*. Before moving to the next phase, we must verify the current phase properly and specifying in a document called *component specification document (CSD*) for future use.

vi. Phase 6: Standard Coding, Unit Testing, Verification and Specification

All the components identified during the last phase may not be available in the component library. The *objective* of this phase is *to write program code for the unmatched components*. Often, a few unmatched components may work as desired just with a suitable added interface. In those cases, the benefit analysis must be done to take the decision whether to develop the interface only or the unmatched components from the scratch. *The unmatched modules must be coded properly following the standard coding guidelines and practices* laid down by the organization itself or the available standard conventions as per the organization interest. These newly developed components must be tested thoroughly since these components are going to be used in several systems at different times. Such testing is called *unit or component testing*. The components taken from the component library together with the newly developed components should be sufficient enough to build the whole system. The newly developed components may be added in the component library for future use if it looks justifiable. After verifying and specifying the phase properly, next phase can be started.

vii. Phase 7: System Building: Component Integration, System Testing, Verification and Specification

Once all the individual *components* are gathered, it's the time to *integrate* these to build the whole system preferably following the bottom up approach. Hence, the *objective* of this phase is *to build the whole system by integrating all the components*. However, it is not necessary that, after integrating the pre-tested components successfully, the integrated system will work correctly. Various types of problems such as type mismatch, number of parameter mismatch, return type mismatch etc. may arise. Hence, there is a need to test the integrated system at different level of integration. This is called *integration testing*. Now, the complete build-up system has to be tested thoroughly using the different testing techniques to check the correctness of system functionality. The testing at this topmost level is termed as *system testing.* After performing the different testing, the corresponding *test report* has to be prepared for use during system validation and maintenance activities. Finally, the phase verification is to be carried out prior moving to the next phase.

viii. Phase 8: System Validation, Verification and Specification

Merry successful verification of the system doesn't ensure the fulfilling of all client requirements! By successful verification of the system, we can only ensure that whatever the functions are implemented in the developed system do work correctly, but does it mean that, all the function required by client are implemented in the system? **No**. The ***objective*** of this phase is *to check* whether *all the functional requirements* as specified in the SRS document specified by the client *are exactly included the system or not.* There must be one to one correspondence between the functions in the SRS document and function supported by the system. Performing this activity is called ***system validation***. Not only the system functionality but also the quality of the system has to be validated. Unlike the other phases, at the end, this phase to be verified and the out come of the system validation activity are to be specified in a document called ***validation report*** and stored for further use.

ix. Phase 9: System Deployment, Implementation and Specification

Once the system is validated, now it's the time to deliver the system to the client and implement the system at client site. Again, some more changes may be required to accommodate and adjust for proper functioning of the system. *Delivering the system to client should not be taken as a formality! Ultimately you- the developers are not going to use the system, but the users definitely.* Until the users are not able to use the system effectively and efficiently, developing the system remains purposeless. We must facilitate the user to understand and feel comfortable with the system at use. There are basically two tools for this purpose. First, the *documentations* and second, *training*. Necessary training has to be provided to all different categories of users within their operational scope. The user refers to the documents to solve problems at any point of time during the system use. The ***objective*** of this phase is *to deliver, implement the system at client's work-site and train the users, if necessary.* After verification of the phase necessary documents are to be prepared and retain.

x. Phase 10: On Site System Testing Verification, and Specification

Although, system testing is completed prior to system implementation, but due to different environmental changes and other reasons, the system may not function correctly at the work-site. Hence, after implementation, the system needs to be tested at work site too. This testing is called ***on-site system testing***. The ***objective*** of this phase is to *check the system performance at work-site*. Finally, the *on-site system testing report* has to be prepared and to be retain after the phase verification. At this point, the current system is at work.

xi. Phase 11: System Maintenance, Verification and Specification

Merry successful system implementation and functioning is not the end job. There is a well saying that *no software is correct at all.* Moreover, Lehman's first law related to software says, *"Software product must change continually or become progressively less useful"* [5]. Software Maintenance denotes any changes made to a software product after it has been delivered to the client. Maintenance is a continuous process over the software life cycle. The ***objective*** of this phase is *to provide the post delivery services to the system for its desirable functioning*. Maintenance support is to be provided to retain and improve the system quality over its lifetime. The maintenance may be of different types i.e. *corrective maintenance, adaptive maintenance, perfective maintenance and preventive maintenance* [5,6]. Finally the *maintenance report* is to be made periodically and kept for future reference.

It should be clear that deliverables from any phase might be given as input to the other phases if needed.

xii. Phase 12: Configuration Management

As we have seen, *system requirements always change during system development and use*. Accordingly, these changes have to be made in associated documents and dependable. Finally, these changes are *to be incorporated into new version of the system.* Hence, it is clear that, the deliverables from different phases are to be maintained for future use. As we have seen over the past discussion that, *from each phase different documents are produced and need to be kept properly for future use.* The patterns are even to be kept in such a way that, on demand we must be able to identify, search, and locate all these components for adaptation. Hence, we need to have an efficient *document and component keeping system*. If there is no proper management and control over these changeable particulars, then it is very tough to incorporate these necessary changes in the following version of the system. *The means by which the process of software development and maintenance is controlled is called configuration management.* The ***objective*** of the configuration management is the development of procedures and standards for cost effective managing and controlling the changes in the evolving software system aiming *to keep track of all the important* deliverables obtained from different phases.

xiii. Phase 13: Project Management

Unlike the configuration management activity, the project management activity has to be carried out in parallel with all the other software development phases. While developing software, we need to carryout some management activities that are part of project management. The ***objective*** of this phase is to *perform the project management activities including project planning, monitoring, controlling, directing, motivating and coordinating.*

X. Analysis of the BRIDGE Model

The in-depth study of the BRIDGE model discloses a lot of information that may be used to analyze the model. These are briefly discussed below:

A. Findings from the Study of BRIDGE Model:

The findings from the BRIDGE model are listed below:

ix. It *involves the client* over the entire development life cycle activities.
x. It keeps continuous *communication with the project management* team.
xi. It explicit *verification of individual* phases.
xii. Separate software *architecture design* phase.
xiii. Separate *system deployment* phase.
xiv. Separate *on-site system testing phase.*
xv. Supports *components based* software development.
xvi. It emphasizes on *standard coding*.
xvii. It considers *configuration management* as a separate activity.
xviii. It forces to *specify* all the phase deliverables.
xix. It explicitly instructs to *validate the system.*

B. Impact analysis of findings from BRIDGE Model Study

In this section, impacts of the findings from BRIDGE model studies on the project goal are analysed distinctly.

i. Impact of continuous client involvement:

It is experienced that, as the system is more studied and analyzed over the time, the client specifies more new requirements. Satisfying these requirements, *client satisfaction* and *software quality* are improved with great impact on both *project* and *organizational goal*. Moreover, involving the client over the entire SDLC *project risks can be alleviate* up to a significant extent. By means of continuous client involvement, this model can embed the *prototyping paradigm of software development.*

ii. Impact of continuous project management team involvement:

The impact of involving the project management team over the SDLC model may facilitate *effective project management activities* such as *project planning, progress monitoring, project controlling*, r*isk management, Motivation* and individual *performance analysis* used for organizational and personal appraisal.

iii. Impact of explicit verification activity:

By verifying the individual phases indirectly the *phase entry and exit criterion* may be satisfied which reduces the error occurrence rate in the later phases. This may even overcome the well-known *99% complete syndrome problem*. Verification helps in *early error detection and correction reducing total development cost having direct impact on software testing, quality control and timely product delivery.*

iv. Impact of software architecture design:

Software architecture is the key framework better project understanding and communication with the various stakeholders. Software architecture has a profound influence on organization functioning and structure [4]. Designing the software architecture has the direct impact on the software quality attributes such as *performance, security, safety, availability, maintainability, scalability, productivity, cost, effort and timely product delivery.*

v. Impact of separate system deployment phase:

It directly maps the *environmental view supported in UML*. There is a very poor practice of considering system delivery as just a formality. Proper training must be given to the users for efficient and effective system use. More over it helps to handle all software crisis related to product deployment improving the software quality.

vi. Impact of separate on-site system testing phase:

The on-site testing helps to improve system quality and client satisfaction reflecting the long-term goal of the Organization.

vii. Impact of component based software design:

The component based software design helps in achieving better *software maintainability, reusability, productivity and quality reducing total development cost and effort.*

viii. Impact of following standard coding:

Following standard coding practices and conventions have remarkable impact on *better understanding* of the code written by others *reducing efforts in error isolation and system testing improving the maintainability, quality of the software. It does encourage good programming practices.*

ix. Impact of configuration management activity:

Configuration management activities improve different documents and components management. It does facilitate component repository and reusability reducing total development cost and efforts improving the software quality and increasing organizational assets simultaneously.

x. Impact of document specification:

The different specified documents facilities *better system understanding* leading to *ease error handling*. These are the

means of communication among teammates and stakeholders. It helps in *reduction of testing and maintenance efforts*.

xi. Impact of system validation:

System validation ensures correct system functionality by error detection achieving the goal of better quality software development. Finally, it increases degree of client satisfaction attaining long-term project and organizational goal.

XI. Validating the BRIDGE Model in Support of Goodness Criterion

The proposed BRIDGE model does satisfy almost all the goodness criterion [2] of a good software development process. In this section, I discuss the supporting issues for validating this model against the individual goodness criteria.

i. Support towards project goal reflection

As per the definition of software engineering given by Stephen Schach [7], the goals of software project are:

a. Developing ***quality*** software.
b. Developing the software ***within budget***.
c. Delivery of the software ***within time***.
d. ***Satisfying customer*** requirements.

By focusing on the phase verification and validation activities, and recommending software testing at different levels, this model reflects the goal of developing *quality software*. Again, specially performing economic feasibility analysis and involving the management over the process, the model reflects the goal of developing the software *within budget*. On stream, by involving the project management team over the entire process development model, this model puts focus on proper management control to follow the time constraints on the project development. Finally, by means of client involvement over the complete software development process, the BRIDGE model achieves the goal of *customer satisfaction*.

ii. Support of Predictability

The software architecture is the best document to predict the different project parameters. Having a separate software architecture phase and risk analysis, this model achieves the predictability criteria.

iii. Support of testability and maintainability

Emphasizing on component based software development and component reusability concept, this model highlights the testability criteria. In addition, designing the software architecture gives the foundation for meeting maintainability criteria with a separate phase related to software maintenance.

iv. Support towards change

Designing software architecture and by supporting maintainability, this model achieves the change management criteria directly with consistent support from configuration management.

v. Support of early defect removal

By involving the customer over the entire development process, it is possible to detect errors at earliest and performing verification activity following each phase ensures early defect detection and removal.

vi. Support of process improvement and feedback

During the configuration management activities, all the prepared documents and reports are stored. Project completion analysis report with the available documents and the reports from configuration database, can be used to judge and identify the activities needing process improvement and applying the same in the next project. Customer comments and recommendations can be used as the feedback for further process improvements.

vii. Support of Quantitative Progress Measurement

Directly, each phase indicates a milestone towards the project completion. All the deliverables from various phases of this process model can be used to measure the progress of the work completed.

viii. Support of Process Tailoring

Since the process activities are decomposed in several phases, at necessity, more than one phase can be combined and any phase can be further decomposed into sub phases or even might be dropped depending on the project characteristics. Hence, it may be concluded that, the BRIDGE model satisfies all the desired characteristics of a good software process model.

XII. Suitability of the BRIDGE Model

This model can be used to both simple systems as well as complex systems. It supports the object oriented, component based software development paradigm. By process tailoring, this model also can be applied to develop any software projects that are directly unfit to the actual model. Hence, the suitability of the BRIDGE model for any modern software development is justified and may be recommended for any kind of software project development.

XIII. Limitations of the BRIDGE Model

Along with the strong suitability, this model has some limitations as pointed down below:

a. Non-considering the implementational issues.
b. Abstracts the different techniques to be used in different phases.
c. Required to be validated by industrial practice.
d. It doesn't consider professionals skill level.
e. The BRIDGE model seems to be complex.

XIV. Naming Significance: BRIDGE

The schematic diagram of the proposed model looks like a bridge. In a bridge, the entire load is on the bridge floor, but this load has distributed over all the pillars for its survivals. Directly the project pressure is on the "Project Management" and this pressure has to be distributed over "Client Interaction", "Configuration management " and other the phases indirectly- the pillars of the model. Keeping this point of view the name, BRIDGE, is given and justified.

XV. Conclusion

After the complete analysis, it can be conclude that if the BRIDGE model is followed to any software project development, most of the software crisis may be overcome up to great extent delivering the fully functional system with better quality within time and budget achieving the true goal of any software project development.

References

[1] Gerg Goth, "*Software-as-a-service: The Spark That Will Change Software Engineering?*" IEEE distributed systems, July 2008.

[2] Jalota P., *Integrated Approach to Software Engineering*, Narosa, Third Edition. 2006.

[3] Jon Holt, *Current practice in software engineering: a survey*, Computing and Control Engineering Journal, 1997.

[4] Joseph F. Maranzano, Sandra A. Rozsypal and Gus H. Zimmerman, Guy W. Warnken and Patricia E. Wirth, *Architecture Reviews: Practice and Experience*, IEEE Software, 2005.

[5] Mall R., *Fundamentals of Software Engineering*, PHI, Second Edition, 2008.

[6] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, Mc-Grawhil, Sixth Edition, 2005.

[7] Stephen Schach, *Software Engineering*, Vanderbilt University, Aksen Association, 1990.

[8] Pfleeger S. L., *Software Engineering: Theory and Practices*, Pearson Education, Second Edition, 2007.