ABSTRACT

A COLLABORATIVE, HIERARCHICAL, INCREMENTAL, AND PROBLEM SOLVING INFORMATION SYSTEMS DEVELOPMENT MODEL

by
Timothy Joseph Burns

The "software crisis" has been a much discussed and debated topic in Information Systems research. A core cause of the crisis is often identified as the methodologies and approaches used to develop information systems. Thus, over the years a multitude of methodologies have emerged in support of quality software. While many of these methodologies have been effective, research has shown that system development is a highly circumstantial process, and that no one methodology can be optimal in every context of every project. It is also a fact that system development practitioners have employed ad hoc approaches to modify formal methodologies in order to create a better fit for their circumstances.

This research aims to present a more formal approach, based on general systems theory, to adapt existing information system development methodologies through the identification of common isomorphic properties. Design science guidelines are applied in the creation of this new model and in its validation. This theoretical model is intended to facilitate the normalization of system development methodologies by means of a framework that can guide the tailoring of methodologies for information systems development.

The functionality and effectiveness of the model proposed here was also evaluated using a "2 x 2 factorial design" experiment. Subjects were assigned to one of

small size and low cost. The present experimental setup consists of a DSP and a Field Programmable Gate Array (FPGA) for parallel processing with the FPGA generating the LTP signal and the DSP performing the signal processing. It is the final objective of this work to implement the entire software system with on a single microcontroller such as the Freescale MPC566.

A COLLABORATIVE, HIERARCHICAL, INCREMENTAL, AND PROBLEM SOLVING INFORMATION SYSTEMS DEVELOPMENT MODEL

by

Timothy Joseph Burns

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Information Systems

Department of Information Systems

January 2007

APPROVAL PAGE

A COLLABORATIVE, HIERARCHICAL, INCREMENTAL, AND PROBLEM SOLVING INFORMATION SYSTEMS DEVELOPMENT MODEL

Timothy Joseph Burns

8/29/06

Dr. Fadi P. Deek, Advisor — Date
Dean, College of Science & Liberal Arts and Professor of Information Systems, New Jersey Institute of Technology

08/29/06

Dr. Vassilka Kirova, Committee Member — Date
Lucent Technologies

8/29/06

Dr. Robert Klashner, Committee Member — Date
Assistant Professor of Information Systems, New Jersey Institute of Technology

8/29/06

Dr. James McHugh, Committee Member — Date
Professor of Computer Science, New Jersey Institute of Technology

8/29/06

Dr. George R. Widmeyer, Committee Member — Date
Associate Professor of Information Systems, New Jersey Institute of Technology

BIOGRAPHICAL SKETCH

Author: Timothy Joseph Burns

Degree: Doctor of Philosophy

Date: January 2007

Undergraduate and Graduate Education:

- Doctor of Philosophy in Information Systems
  New Jersey Institute of Technology, Newark, New Jersey, 2007

- Master of Business Administration (Honors)
  Iona College, New Rochelle, New York, 1994

- Bachelor of Applied Science
  Florida Atlantic University, Boca Raton, Florida, 1982

Major: Information Systems

Presentations and Publications:

Burns, T., Klashner, R. , & Deek, F. P. , (2006) "An Empirical Investigation of a General System Development Model", Proceedings of the Twelfth Americas Conference on Information Systems, Acapulco, Mexico, August 2006.

Burns, T., Klashner, R., (2005) "A Cross-Collegiate Analysis of Software Development Course Content", ACM SIGITE Conference, Newark, NJ, October 2005.

Klashner, R., Burns, T., (2005) "A Method to Refine and Tailor System Development Methodologies", Proceedings of the Fourth Annual SIGSAND Symposium, Cincinnati, OH, April 2005.

To my family, especially:

My parents, Dr. James and Alberta Burns; you taught me the importance of an education and inspired this undertaking.

My children, TJ, Aly, and Andrew; I hope that I have passed to you that importance of an education and that this document will serve as a symbol to you that through hard work and perseverance you too can someday attain your dreams.

My wife, Patti; No dream would be complete without you…

ACKNOWLEDGEMENTS

I would especially like to thank my advisor, Dr. Fadi P. Deek, who was so instrumental in clearing obstacles from my path. Fadi is an incredible person and I will always be grateful to him for his wisdom, advice, and support. I would also like to thank Dr. Robb Klashner, who initially worked with me and challenged me to constantly improve. Together, Dr. Deek and Dr. Klashner invested a tremendous amount of time and effort in me and helped me to realize my potential as an academic.

Thank you to my committee members; Dr. Vassilka Kirova, Dr. James McHugh, and Dr. George Widmeyer for their insights and suggestions. Particularly to Dr. Widmeyer for the suggestions on Design Science Theory and Dr. Kirova for her help with the experiment.

Thank you to all of my colleagues at Ramapo College of New Jersey for their encouragement and support, particularly Professor Stephen Klein, Dr. Yuan Gao, Dr. Alexander Vengerov, and Professor Cherie Sherman for their suggestions on improving this research.

Thank you to the Information Systems community of NJIT, particularly Dr. Roxanne Hiltz, Dr. Murray Turoff, and Dr. Michael Beiber for providing insight and inspiration along the way.

Thank you to all of the instructors who allowed me to use their classes as subjects for the experiment. This includes Dr. Vassilka Kirova, Joan Kettering, Osama Eljabiri, and Morgan Benton from NJIT and Professor Stephen Klein from Ramapo College of New Jersey.

Thank you to the students who chose to participate in this research as subjects. I am grateful to you for your time, effort, and diligence in performing the necessary tasks so that I could collect the data for this research.

I would especially like to thank the MBA students from Ramapo College of New Jersey who acted as expert judges for the experiment. This includes Robin Keller, Elizabeth Bein-O'Brien, Jaclyn Petrin, and Graziella Vazquez. Robin and Liz, in particular, went over and above the call of duty in their participation of this research. I certainly could not have completed this without them and I will be forever grateful.

I would like to thank family and friends who were a constant source of encouragement, especially my mother Alberta, my brothers Mike and Matt, my mother-in-law Eileen and, all of my brothers and sisters-in-law.

Last, but not least, I would like to thank my immediate family who allowed me the time to complete this endeavor. In particular, my wife Patti, whose wisdom and rock solid support kept me focused and sane throughout this process and whose unwavering love gave me the strength to persevere.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF TABLES
(Continued)

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1
INTRODUCTION AND BACKGROUND

1.1 Introduction

This research is motivated by the current state of software development projects. There is little doubt that there is a software crisis (Brooks 1987). According to a study of 280,000 development projects conducted by the research firm "The Standish Group" in 2000, twenty-three percent of all software development projects are cancelled before they finish and forty-nine percent of software development projects are completed late, over budget, or do not meet the requirements of the users and/or the organization in which they are implemented. Only about one in four (twenty-eight percent) of all software development projects are considered successful (SOFTWAREmag.com 2001). Studies in prior years have also reported similar results (Whiting 1998, Standishgroup.com 1994).

Research has also shown that there is a divide between the IS development practices being developed through academic research and those used by practitioners (Fitzgerald 1997, Fitzgerald, Russo, O'Kane 2003, Burns and Klashner 2005). There is a plethora of IS development methodologies, many of which claim to be the "silver bullet" (Brooks 1987) to the software crisis. In many cases they have the potential to be the silver bullets if used in the right environment. However, research has shown that none is the silver bullet all the time (Cockburn 2002, Fitzgerald et al. 2003).

The general field of study for this dissertation is Information Systems (IS). The sub-class will be IS Development and the specific research topic will be in the realm of IS Development Methodologies. The scope will be limited to presenting evidence that demonstrates why this research was undertaken, the methodology employed in

conducting the research, and the results of that research. The results are analyzed and discussed and conclusions are drawn.

This dissertation is presented in several chapters. Chapter 1 introduces and presents the research problem. This includes the background of the problem, related research, and a detailed statement of the problem. Chapter 2 provides a literature review and state of the art analysis of IS development methodologies. It is organized in a rough chronological order that starts with original IS development methodologies and leads up to current practices and research areas. It shows how each methodology has it own innate advantages and disadvantages and how each can be the optimum methodology given the appropriate environment and project contingencies.

Chapter 3 introduces the collaborative, hierarchical, incremental, and problem-solving model (CHIPS). It demonstrates the theoretical foundation for the model and then gives a detailed description of the model. Chapter 3 then details how the research contributes to the field of study and who will benefit from the research. The chapter ends by posing the research questions studied.

Chapter 4 presents the methodology employed in this research. It explains the experiment that was conducted and its theoretical basis. It lists the hypotheses that were examined. It also lists the instrument, tools and methods that were used to gather and analyze the data.

Chapter 5 presents a statistical analysis of the experiment data using appropriate descriptive and inferential statistical measures. Chapter 6 is a discussion of those results. Chapter 6 also sums up this work. It lists the scopes, limitations, and challenges of the project. It includes a final concluding statement and implications for future work.

1.2 Background

"A methodology is a recommended collection of phases, procedures, rules, techniques, tools, documentation, management, and training used to develop a system" (Avison and Fitzgerald 2003). There are hundreds of software development methodologies (Fitzgerald et al. 2003). Table 1.1 summarizes some of the most common methodologies and approaches to system development.

IS development methodologies can be classified several ways (Avison and Fitzgerald 2003, Fowler 2003). Examples are by methodological era, by approach, or by degree of agility. Software development methodologies have progressed through several distinct eras (Avison and Fitzgerald 2003). The methodological era approach would categorize the initial era as a period when no methodologies were used. During this period, hardware limitations placed limits on the size and complexity of software systems allowing systems to be effectively developed without methodologies.

The advent of more powerful computers in the 1960's and 1970's was a precursor to more sophisticated and complex software systems. This era marked an early methodology period where a software development life cycle (SDLC) such as the waterfall approach (Royce 1970) was predominant (Avison and Fitzgerald 2003). Under this approach, the project is divided into several distinct stages to be completed in a linear progression. While the waterfall approach was a significant step in formalizing software development, it soon became apparent that there were many inherent shortcomings to this methodology. Chief among the shortcomings is that the lack of flexibility can allow design flaws to not be discovered until late in the development cycle (Racoon 1997).

Changes made late in the development cycle are often more costly (Beck 2000). Another shortcoming is the lack of planning for future changes in the requirements of the software, which are almost always present (Highsmith 2000).

In response to the shortcomings of the traditional SDLC, the methodology era was born. During this era, many new methodologies were introduced. Methodologies from this era can be classified according to approach. Significant approaches include structured, data-oriented, process-oriented, prototyping, participative, object-oriented, and systems (Avison and Fitzgerald 2003).

Many people argue that in recent years we have entered a post-methodology era (Avison and Fitzgerald 2003, Fowler 2002). This era is marked by researchers and practitioners questioning the philosophies of the methodologies from the methodology era. The most serious criticism of these methodologies is that they are bureaucratic and labor intensive (Fowler 2002). For this reason, these earlier methodologies have recently been classified as the "heavy" methodologies (Fowler 2002). In response to this, several new methodologies have been introduced in this post-methodology period. These new methodologies are referred to as lightweight or agile methodologies because of the lack of bureaucratic contingencies inherent to the heavy methodologies (Fowler 2002). These agile methodologies are considered by many to be "a" methodological (i.e., a negative construct that connotes an open set of attributes that are essentially not methodical) (Truex, Baskerville, and Travis 2000). They are distinguished by their emphasis on adjusting to the project environment through adaptation and have less emphasis on prescribing a plan to follow (Fowler 2002). The biggest criticism of the agile methodologies has been the lack of empirical evidence supporting the claims of their

benefits and their lack of theoretical foundation (Abrahamsson, Warsta, Siponen, and Ronkainen 2003). However, there is a growing body of literature both supporting and repudiating the claims of success of the agile methodologies (Abrahamsson et al. 2003, Choi and Deek 2002).

Table 1.1 Summary of Popular Software Development Methodologies/Approaches

Methodology	Author	Type	Characteristics	Advantages	Disadvantages
Waterfall	Royce	Heavy	Project is divided into distinct phases	Ability to quantify an abstract subject. Especially useful to management.	Costly if req. change. No product until late in the process
Prototype	?	Heavy	Develop an initial model of the system for user feedback	Allows the user to provide feedback before a large investment made	Lack of documentation and poor project visibility
Spiral	Boehm	Heavy	Successive refinement of requirements and design	Incorporates risk	Can incur endless fiddling
Joint Application Development	IBM	Heavy	Workshops where users and I.S. people meet to define system	Brings important participants of a project together to hash out the details	High Cost and time consuming
Rapid Application Development	Martin	Heavy	Development life cycle designed for faster development and higher quality	Quicker development and less development costs	Higher risk of low quality systems.
Object Oriented	Coad, Yourdon, Robinson, Booch	Heavy	Decompose a problem into objects that encapsulate data and behavior	Better modeling, analysis & design. Improved communication	Increased development time..
Extreme Programming (XP)	Jeffries Beck	Agile	Short cycles, incremental planning, flexible schedule	Increased customer involvement, teamwork, & communication	Code centered vs. design centered. Lack of documentation
Crystal	Cockburn	Agile	People-centric, ultra-light, shrink to fit	Takes into account current cultural conditions	Requires constant fine tuning.
Scrum	Schwaber	Agile	Lightweight, iterative, incremental process	Increased productivity and adaptability	Poor resource estimating. Limited to small teams.
Feature Driven Development	Deluca Coad	Agile	Design and Build by feature Model-driven short-iteration process.	More planning then rest of agile methodologies	Increased complexity
DSDM	DSDM Consort	Agile	Works under time constraints of RAD	Increased user satisfaction. RAD time and cost savings	Requires special training to use (not free). Complex.
Adaptive	Highsmith	Agile	Based on complex adaptive systems. Emergence vs. determinism	Adaptive to change.	Poor resource estimating.
Open Source	Torvalds	?????	Collaborative Development	Increased quality and reliability.	Adapting the process to business/commercial applications

1.3 Related Research

Every software development project is unique in terms of culture, people, length, etc. (Cockburn 2002). For this reason, methodologies should be "custom fit" to a project depending on contingencies (Cockburn 2002, Fitzgerald et al. 2003). Research has also shown that no one software methodology is best in all circumstances (Cockburn 2002, Fitzgerald et al. 2003). Circumstances may even change during the life of a project that could affect the methodology that should be used at a particular phase of the project.

There are two accepted models for tailoring methodologies to the contingencies of a project (Brinkkemper 1996, Fitzgerald et al. 2003). The contingency factors approach suggests that specific features of the development context should be used to select an appropriate methodology from a portfolio of methodologies (Fitzgerald et al. 2003). The main problem with this approach is that developers are not familiar with every methodology and may not be familiar with the one that best suits the situation (Fitzgerald et al. 2003).

Method engineering is when software developers build a meta-method that is made up of method fragments from popular development methodologies (Brinkkemper 1996, Fitzgerald et al. 2003). Method fragments are coherent pieces of IS development methods (Brinkkemper 1996). Method fragments can be distinguished into process and product method fragments. Product fragments model the structure of the products (deliverables, diagrams, tables, models), while process fragments are models of development processes (project strategies and detailed procedures) (Brinkkemper 1996).

The method fragments are stored and selected from a repository of all available method fragments. The fragments are each designed to handle a particular contingency

inherent to the software project. By combining the method fragments into a meta-method, a methodology can be created that is custom fit to the project

Figure 1.1 shows a typical method-engineering model. A repository holds all the method fragments. The developers use a method engineering tool to pick the fragments from the repository to build the meta-model.

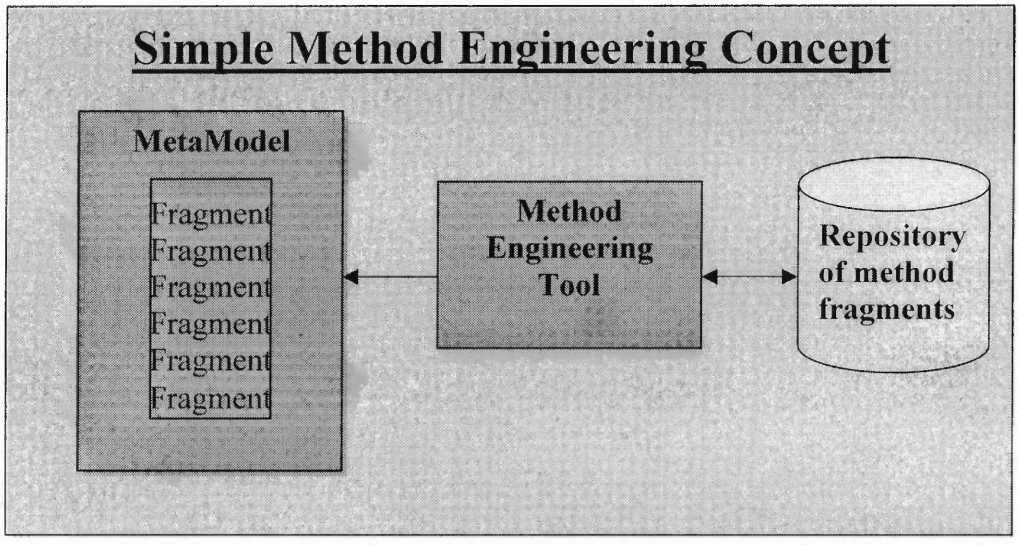


Figure 1.1 A typical method engineering concept.

Method engineering is implemented via a tool. The typical method-engineering tool uses a method engineering language (MEL) to describe and manipulate method fragments (Brinkkemper 1996). In essence, the MEL, through an analyst, is used to arrange methodology fragments in the optimum configuration based on several criteria. In traditional method-engineering theory, the project then proceeds to follow the methodology created, or at least until a milestone or task is completed. This process may be repeated several times throughout the life cycle of the project.

Method engineering (ME), while a promising approach, has several shortcomings:

1. It is impossible to plan for every contingency that may come up during the development of the software product and therefore critical fragments will always be missing (Rossi 2000).

2. The repository of pre-defined method fragments is somewhat static, and can only be changed by the developers of the ME tool.

3. The combination of problems 1 and 2 give the typical ME tool a prescriptive nature, rather than an adaptive one. The tool does not adapt well to unforeseen circumstances.

4. The burden of selecting the correct fragment falls upon the analyst (even though the selection is aided by the ME tool) (Truex et al. 2000).

5. ME can be a complex procedure that requires implementation via a CASE tool and thus be problematic (Fitzgerald et al. 2003).

The area of method engineering, by its nature, has links to many other research areas including project management, software configuration management, software engineering environments, software process modeling, and computer supported cooperative work (Brinkkemper 1996). However, it is problem solving that is at the heart of software development (Highsmith 2000). Problem solving is also the key to software adaptation (Highsmith 2000). In fact, adaptation and problem solving are in many ways synonymous in the software development realm. Therefore, a project that is developed from a pure problem solving standpoint, is in essence, one that is developed in a purely adaptive or emergent fashion. The problem with developing from a purely problem solving, adaptive philosophy is that it fails to take advantage of the known and "orderly" parts of the project environment (Highsmith 2000). In other words, the ideal situation is one where we plan for what we can, but at the same time, have a problem solving mechanism in place to facilitate adaptation to unforeseen circumstances.

Early problem solving methods were generally divided into two camps (Deek, Turoff, McHugh 1999). The first camp advocated using the traditional scientific method to solve problems (Dewey 1910), while the second camp believed in a creative, non-systematic problem solving methodology (Wallas 1926, Hadamard 1945, Poincare 1913). Later methods such as Polya (1957) were able to combine elements from both of these two approaches (Deek et al. 1999).

One of the primary contributors to the field of problem solving was Polya (Deek et al. 1999). In his book, "How to Solve It", Polya lays out a four part process to problem solving (Polya 1957). The steps of the process can be summarized as follows:

1. Understand the problem - This step includes defining the data, the unknown, and the condition. It also includes separating the parts of the condition and using figures to illustrate the problem.

2. Devise a plan – This step involves finding the connection between the data and the unknown. You may have to look at similar problems, or parts of the problem to find a solution.

3. Carry out the plan – This step involves implementation and verification of each step.

4. Examine the solution - This step involves checking the solution to prove that it was indeed correct. It also involves checking to see if the solution obtained can be used to solve other problems.

Polya's work is generally the most often cited in the field of problem solving. However, there has been significant research in the field. Table 1.2 presents a summary of several problem-solving methods (Deek et al. 1999). Table 1.2 characterizes the models using Polya's basic format. Each model (albeit in its own manner), follows some semblance of Polya's process in that they understand/define the problem, plan a solution, implement the solution, and verify the results.

A common attribute of problem solving methods is the implied and often times explicit attribute of hierarchy. Many problem-solving methods involve a process of defining a problem in terms of its subordinate problems. Polya and others recommend decomposing the problem into smaller problems (Polya 1957, Chandrasekaran 1990). Problem decomposition is a primary problem solving method in computer science (Ginat 2002).

Table 1.2 Summary of Problem Solving Models (From Deek et al. 1999)

	Understanding and Defining the Problem	Planning the Solution	Designing and Implementing the Solution	Verifying and Presenting the Results
Dewey (1910)	Define Problem	Suggest Possible Solutions	Reason About Solutions	Test and Prove
Wallas (1926)	Preparation	Incubation Illumination		Verification
Polya (1945 & 1962)	Understand Problem	Devise Plan	Carry out Plan	Look Back
Johnson (1955)	Preparation	Production		Judgment
Kingsley & Garry (1957)	Clarify and Represent Problem	Search for Clues Evaluate Alternatives	Accept an Alternative	Test Solution
Osborn & Parnes (1953 & 1967)	Find Facts Find Problem	Find Idea	Find Solution	Find Acceptance
Simon (1960)	Intelligence	Choice	Design Implementation	
Rubinstein (1975)	Get Total Picture	Withhold Judgment Model	Change Representation Ask Questions	Doubt Results
Stepien, Gallagher & Workman (1993)	Analyze Problem List what is Known Develop Problem Statement	List what is Needed List Possible Actions	Analyze Information	Present Findings
Etter (1995)	Define Problem Gather Information	Generate and Evaluate Potential Solutions	Refine and Implement Solution	Verify and Test Solution
Meier, Hovde & Meier (1996)	Define Problem Assess Situation	Plan Strategy	Implement Plan	Communicate Results
Hartman (1996)	Identify and Define Problem	Diagram Problem Recall Content Explore Alternative Strategies	Apply Content and Strategies Monitor Work-in-Progress	Assess Solution Product and Process

The predominant means of decomposition in software development is the "top-down" approach (while several others do exist) (Ginat 2002). This approach involves taking a problem and decomposing it into subordinate problems. Once a problem has been decomposed, then the subordinate problems are decomposed and the process

continues to repeat itself until the problem has been fully decomposed into a hierarchy of problems. The individual problems are then solved.

This idea of decomposition was also expressed by Chandrasekaran (1990) as "design by decomposition." Chandrasekaran proposed a three step process:

1. Decompose a problem into sub-problems
2. Solve the sub-problems
3. Recompose the sub-solutions

Another research area related to this research is collaborative software development. It has been shown that collaborative problem solving and software development improves the software development process (Defranco-Tommarello and Deek 2002). Experimentation has shown that developers considered people their most used asset in developing software (Defranco-Tommarello et al. 2002). The success of collaborative open source development has been shown through the development of software such as the Linux operating system and the Apache web server (Feller and Fitzgerald 2000, Augustin, Bressler, and Smith 2002).

1.4 Detailed Statement of the Problem

While the search continues for the best methodology (whether heavy or agile), recent research has pointed out several distinct conclusions:

1. No one method can claim to be the best methodology for every software project (Cockburn 2002, Fitzgerald et al. 2003).

2. Traditional development methodologies tend to be highly bureaucratic and labor intensive (Fowler 2002).

3. Agile methodologies tend to not offer adequate support for project management (Abrahamsson et al. 2003).

4. Present day practitioners are becoming increasingly discouraged with the traditional methodologies and their shortcomings (Avison and Fitzgerald 2003).

5. Those developers that have continued to use the traditional methodologies have modified and adapted those methodologies to meet the specificities of the project. There is a strong discrepancy between the theory of the methodologies and the way they are applied by practitioners (Fitzgerald 1997, Fitzgerald et al. 2003).

6. Method tailoring approaches (i.e., the contingency factors approach and method engineering) have considerable shortcomings and have not been proven to be effective in their practical application (Fitzgerald et al. 2003).

CHAPTER 2
INFORMATION SYSTEMS DEVELOPMENT METHODOLOGIES

2.1 The Software Crisis

Since the advent of the computer, there has been software development. However, it was not until the 1960's that the combination of affordable, programmable computers and the introduction of third generation programming languages allowed for the creation of large software systems. By the late 1960's, people were realizing that there was a problem, or what some were calling a crisis.

"This software crisis resulted directly from the introduction of third generation computer hardware. These machines were orders of magnitude more powerful than second-generation machines. Their power made hitherto unrealizable applications a feasible proposition. The implementation of these applications required large software systems to be built... Techniques applicable to small systems could not be scaled up... Software development was in a crisis... New techniques and methods were needed to control the complexity inherent in large software systems" (Sommerville 1996).

To address this problem, conferences that were sponsored by NATO, were held in 1968 and 1969. At these conferences, the idea was put forth that software development needed to adopt many of the techniques used in the engineering disciplines and the term "Software Engineering" was coined. At the first major conference on software engineering, Dr. F.L. Bauer said: "What is needed is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines" (Naur and Randell 1969).

This crisis and the solutions there of, led to the development of several software methodologies grounded in their engineering philosophies. "These methodologies have been around for a long time. They've not been noticeable for being terribly successful. They are even less noted for being popular. The most frequent criticism of these methodologies is that they are bureaucratic. There's so much stuff to do to follow the methodology that the whole pace of development slows down. Hence they are often referred to as heavy methodologies..." (Fowler 2002).

In the next few pages, several of the more popular "heavy" methodologies will be described.

2.2 The Waterfall Approach

The first explicit model of the software development process was derived from engineering processes (Royce 1970). This model was called the "Systems Development Life Cycle" or SDLC. It is also well-known as the "Waterfall Model", because of the cascade from one phase to another (Sommerville 1996). The Waterfall Method is depicted in Figure 2.1.

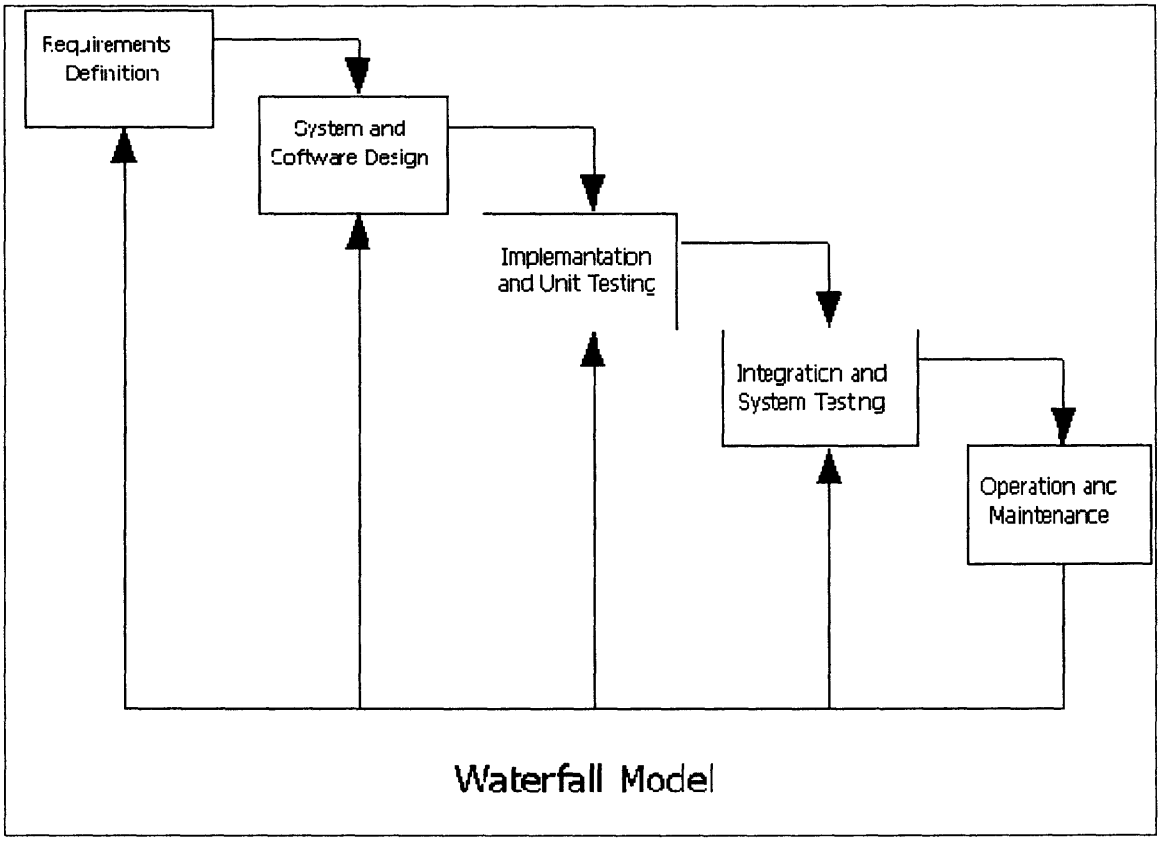


Figure 2.1 The waterfall approach.

The waterfall approach requires that a system development project be divided into several distinct phases or stages. The number of phases or stages can vary, as well as the exact names given to each phase. "Every textbook author and information system

development organization uses a slightly different life cycle model, with anywhere from three to almost twenty identifiable phases" (Hoffer, George, and Valacich 1999).

Each phase of the model produces some sort of deliverable that is reviewed, validated, and signed off on. The phases are generally completed in a sequential, ordered manner, but not necessarily. "Although any life cycle appears at first glance to be a sequentially ordered set of phases, it is not. The specific steps and their sequence are meant to be adapted as required for a project..." (Hoffer et al. 1999).

Some stages can be worked on in parallel and some may be revisited in a later stage of the project. Re-visiting is generally difficult due to the contingent characteristics of subsequent phases. Re-working an earlier phase may cause subsequent phases to be re-worked as well.

Figure 2.1 shows the usual phases associated with the Waterfall model. They are defined as:

- Requirements Definition – formulation of the system specifications and goals
- System and Software Design – establishing the overall architecture and expressing functions in programming terms
- Implementation and Unit Testing – coding and debugging the modules
- Integration and System Testing – integrating the modules and testing the system
- Operation and Maintenance – revising and enhancing the product, correcting any new errors after the product is delivered.

The main strength of the Waterfall method is its ability to quantify an abstract subject. This ability to quantify is especially useful to management, as they are able to see the progress a project is making. Each phase is defined by a set of functions, goals, milestones, and deliverables, making the process highly visible and the project easier to

track. Also, since requirements and specifications are determined at the outset, the project manager is better able to determine his resource needs and establish schedules.

The Waterfall methodology has been under much scrutiny for a number of years. Out of that, four main weaknesses of this model have been identified. First, is the converse effect of its rigidity. Although, it can be beneficial in quantifying a project, it can also seriously detract from the project. In an effort to adhere to the constraints of this model, the development process often allows flaws to remain in effect or fails to plan for future change (Racoon 1997). At a later time, it is costly to revisit a phase and make a change, because other dependencies may have been generated from the phase. To correct an original flaw, several other subsequent phases may have to be revisited. Thus, flaws remain intact. The later you are in the development life cycle, the higher the cost becomes to make a change (Beck 2000). Therefore, the Waterfall method can be costly if requirements change. Consequently, this model is best used on projects where the requirements are clearly defined and there is little risk of change (Boehm 1999).

The second weakness in the Waterfall methodology is the fact that the user does not get to see the working product until late into the life cycle. Although the project is producing documentation and recording milestones along the way, the user does not actually have a hands-on experience with the system until the testing phase. As stated previously, requirement changes become costlier later in the system development life cycle. Thus, by the time the user gets to review the product, it can be costly to fix if there is a problem.

The third main problem with the Waterfall model is its linear nature. Large software projects often times are made up of several "mini" projects or modules. These

modules may all have different timelines. Some modules may complete a phase very quickly, while others may not. Waiting for all of the modules to complete a phase, before moving on to the next phase as a group, can cause inefficient lags and downtime for the developers. This linear process can result in increased costs and also makes reiteration of earlier phases difficult and expensive.

The fourth problem with the Waterfall model is the sometimes wide conceptual gap between the functions defined in the design process and the functions defined in most conventional programming languages. This gap makes translating the original design specifications into computer programs highly complex and often results in confusing, complex, and low readability programs.

While it appears that the list of disadvantages far outweighs the advantages of using the Waterfall method, it still remains a very popular choice among software developers. "Nevertheless, the waterfall model reflects engineering practice. Consequently, it is likely that software process models based on this approach will remain the norm for large hardware-software systems development" (Sommerville 1996).

2.3. Prototyping

Prototyping specifically addresses the second weakness and ultimately the first weakness of the Waterfall model, in that users get to see what the product looks like very early on in the development process (Swift 1989). Prototyping is based on the idea of developing an initial implementation of the system for user feedback, and then refining this prototype through many versions until a satisfactory system emerges. The user can try out the system; albeit a (sub) system of what will be the final product. Trying the system out early allows the user to provide feedback before a large investment has been made in the development of the wrong system. It also reduces the odds that an earlier phase will have to be revisited as the requirements are more clearly defined.

Generally prototyping is split into two categories:

- Exploratory or Evolutionary programming- where the objective is to work with the user to explore their requirements and deliver a final system. It starts with the parts of the system which are understood, and then evolves as the user proposes new features (Boehm 1988).
- Throw-away prototyping- where the objective is to understand the users' requirements and develop a better requirements definition for the system.

Often times, Throw Away prototyping is used in conjunction with the Waterfall mode (Swift 1989). The initial prototype is developed in order to define the requirements of the system and to let the user try out the system before a huge investment is made in the project. The prototype is then thrown away and the traditional Waterfall model is followed for the remainder of the project. This combined methodology can be very effective in that it reduces the risk of encountering the aforementioned risks of the Waterfall method as well of the usual shortcomings of prototyping. These include lack of documentation and low project visibility (Swift 1989).

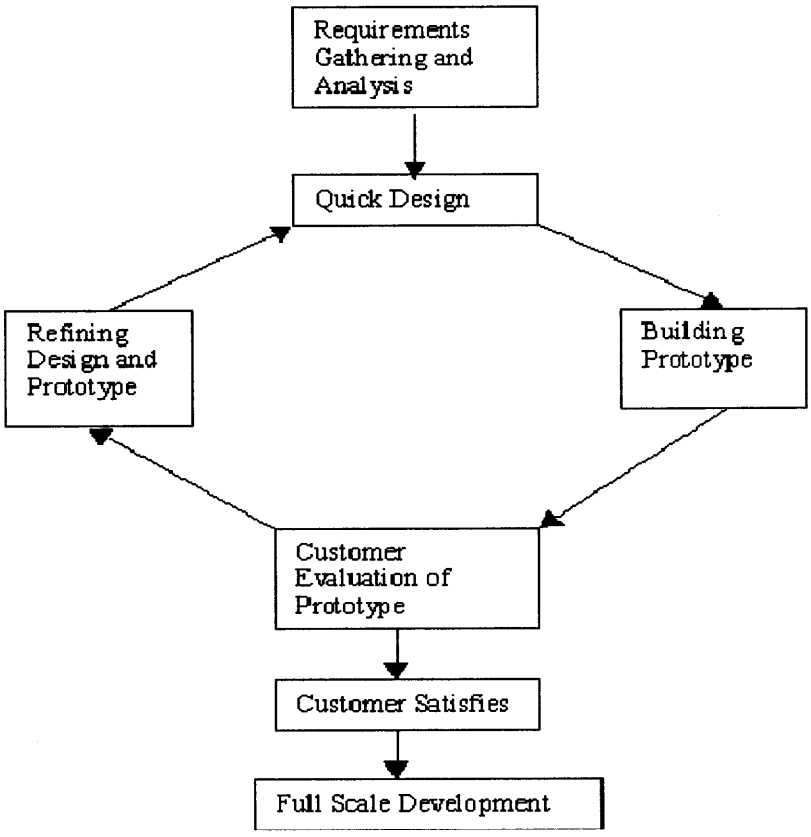


Figure 2.2 Prototype model.
Source: Swift 1989

2.4. Boehm's Spiral Method

The spiral model was developed to address the problems associated with the traditional waterfall methodology. Primarily, it addresses the problem of inherent risk that is ignored in the waterfall model (Boehm 1988). The spiral model recognizes that the process of design will reveal problems with the requirements definition and the process of testing will reveal problems with the design. It plans for these occurrences and allows for the successive refinement of the requirements definition and design to proceed as the software is constructed.

The model consists of four stages, which are reiterated until the project is complete. They are defined as:

- Plan the next stage and what model will be used in that cycle.
- Determine the objectives, alternatives and constraints.
- Evaluate the alternatives, identify and resolve the risks.
- Develop and verify a product.

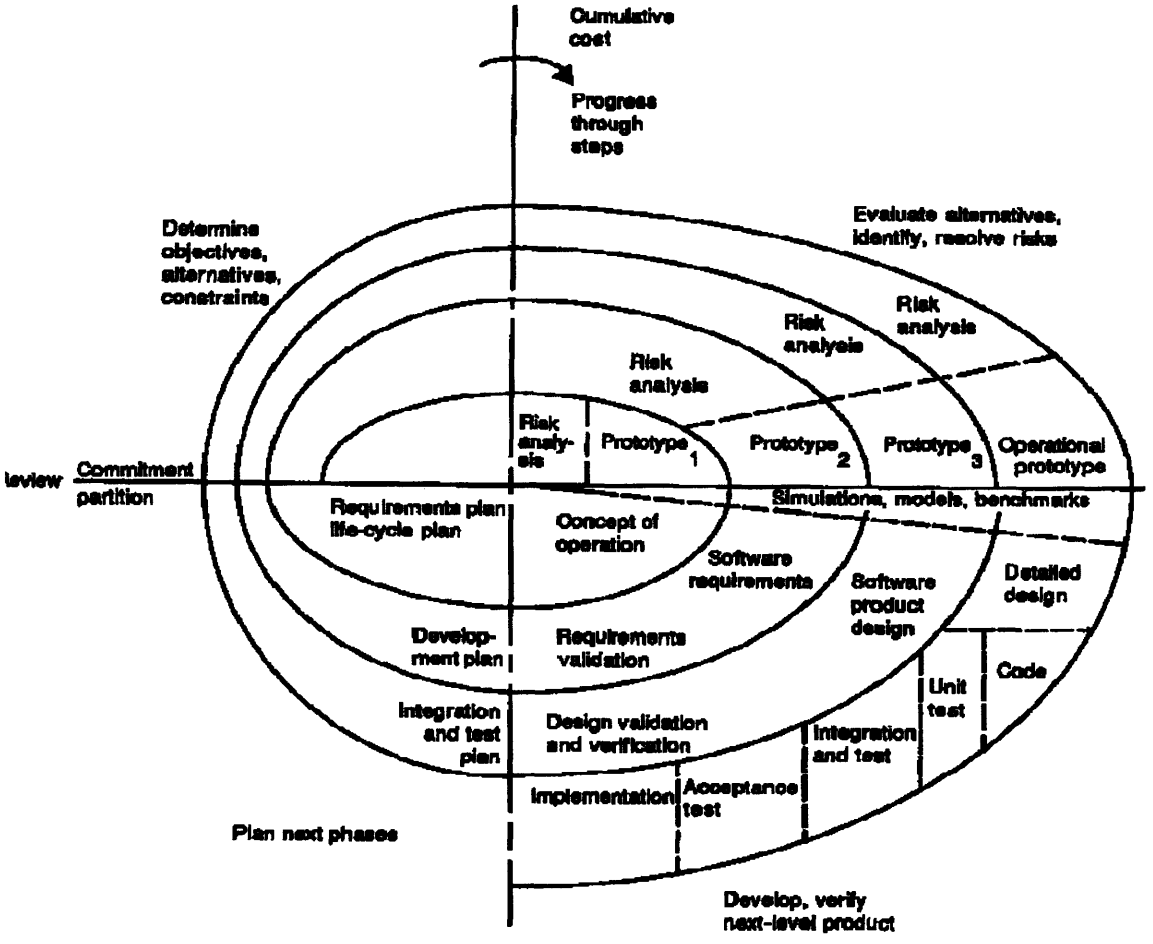


Figure 2.3 The spiral model.
Source: Boehm 1988

The spiral model is a meta-model, for it can incorporate any other model in it (Boehm 1988). Every cycle of stages can be done with a different model, according to what is best. Usually in the first stage of the cycle (the planning stage), it is determined which of the remaining functions left in the system development process have the highest risk. It is these functions that are then tackled in that cycle.

The nature of the spiral model lends itself to be especially useful in developing systems that are at a high risk. Those would be systems that are difficult to define or that have a high probability of future change. This inherent risk factoring is the strength of

the spiral model. Earlier tasks that have been completed can be re-visited without an effort. It is assumed that earlier tasks *will* change and that the development team should be prepared for those changes.

The major problem with the spiral method is that it can become a justification for endless fiddling with the system development process. Development can get stuck in an endless loop, where certain tasks such as program coding or requirements design are done over an over again. Endless fiddling can also lead to the project never ending or never reaching a certain goal. One way to prevent this problem is by making sure that clear goals and mileposts are defined for the overall system. Also, a review phase should take place at the end of each cycle. In this review phase it must be determined if the development process is stuck in an endless iteration, and if it is time to move on and close the book on some prior activities.

2.5 Joint Application Development

"JAD (for Joint Application Development) centers on a three to five day workshop that brings together business area people (users) and IS (Information Systems) professionals. Under the direction of a facilitator, these people define anything from high-level strategic plans to detailed system specifications. The products of the workshop can include definitions of business processes, prototypes, data models, and so on" (Wood and Silver 1995).

Originally called Joint Application *Design,* this process was developed by IBM in the late 1970's to collect information systems requirements and review system designs (Hoffer et al.). It is still used in this capacity by many practitioners (Hoffer et al.). However, others have expanded JAD's role into other phases of the SDLC (Botkin 1994, Wood 1995). JAD is oftentimes combined with prototyping (Botkin 1994) and Rapid Application Development (Bayer and Highsmith 1994, Howard 2002).

JAD participants typically include (Botkin 1994):

1. FACILITATOR: Chairs the meeting and directs traffic, by keeping the group on the meeting agenda. The facilitator is responsible for identifying those issues that can be solved as part of the meeting and those which need to be assigned at the end of the meeting for follow-up investigation and resolution. The facilitator serves the participants and does not contribute information to the meeting.

2. MODELER: Documents the proceedings of the meeting and captures the input as stated by the participants, avoiding paraphrasing and filtering of this information as much as possible. The modeler records the proceedings of the meeting and does not contribute information to the meeting.

3. PARTICIPANTS: Customers in the business area directly or indirectly being affected by this project, who are experts in their field and can make decisions about their work. They are the source of the input to the session.

4. OBSERVERS: Generally members of the application development team assigned to the project. They are to sit behind the participants and are to silently observe the proceedings.

5. PROJECT LEAD: Generally the leader of the application development team answers questions about the project regarding scope, time, coordination issues and resources. They may contribute to the sessions as long as they do not inhibit the participants.

The five phases of JAD are defined by Wood et al. as:

1. JAD project definition.
2. Research on user requirement.
3. Preparation for the JAD session.
4. Conducting and facilitating the JAD session itself.
5. Predicting and obtaining approval of the final document that incorporates all decisions made.

Preparation is the key to JAD, as three of the five phases are carried out prior to the group sessions. During the actual JAD sessions, special rooms may be used and CASE tools may be employed (Hoffer et al. 1999.)

The benefits of JAD are found primarily in its ability to bring all the important participants of a project together to hash out the details of the project. This collaboration can lead to increased understanding of the goals of the project and realize a savings in time needed to complete the project (Hoffer et al. 1999, Wood 1995).

The primary disadvantage of Joint Application Development is the cost. It may get expensive to find special facilities to handle the JAD sessions and to transport all the participants to the JAD location if not all are local. Also, the time of the participants must be considered if the JAD sessions are held in an iterative fashion and must be attended multiple times (Hoffer et al.).

2.6 Rapid Application Development

"While no universal definition of RAD exists, it can be characterized in two ways: as a methodology prescribing certain phases in software development (similar in principle to the spiral, iterative models of software construction), and as a class of tools that allow for speedy object development, graphical user interfaces, and reusable code for client/server applications. Indeed, the tools and methodology are inextricably linked: the tools enable the methodology and circumscribe what is accomplished during a development project"(Agarwal, Prasad, Tanniru, and Lynch 2000).

The term RAD (for rapid application development) is credited to James Martin, who wrote a book on the subject in 1991. "RAD refers to a development life cycle designed to give much faster development and higher quality results than the traditional life cycle. It is designed to take maximum advantage of powerful development software that has evolved recently" (Martin 1991).

Martin suggests that there are four components to RAD. The first consists of software development tools. The other components are people, who must be trained in the right skills, a coherent methodology, which spells out the proper tasks to be done in the proper order; and the support and facilitation of management (Martin 1991).

RAD is similar to JAD in that they both depend on extensive user involvement. "End users are involved from the beginning of the development process, where they participate in application planning; through requirements determination, where they work with analysts in system prototyping; and design and implementation, where they work with system developers to validate final elements of the system's design" (Hoffer et al. 1999).

RAD can often times be characterized as a combining of the JAD and evolutionary prototyping methodologies as depicted in Figure 2.4 (Maner 1997).

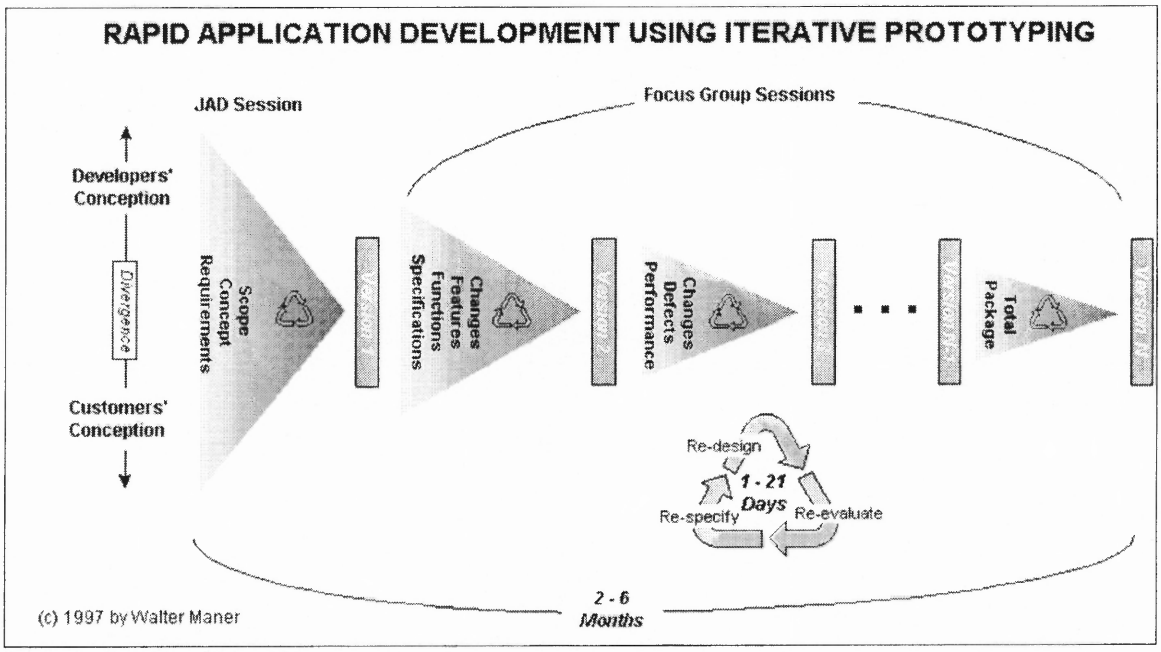


Figure 2.4 Rapid application development.
Source: Maner 1997

Table 2.1 Characteristics of RAD (Maner 1997)

CHARACTERISTICS OF RAD (Maner 1997)

RAD USES HYBRID TEAMS

- Teams should consist of about 6 people, including both developers and full-time users of the system plus anyone else who has a stake in the requirements.
- Developers chosen for RAD teams should be multi-talented "renaissance" people who are analysts, designers and programmers all rolled into one.

RAD USES SPECIALIZED TOOLS THAT SUPPORT:

- "visual" development
- creation of fake prototypes (pure simulations)
- creation of working prototypes
- multiple languages
- team scheduling
- teamwork and collaboration
- use of reusable components
- use of standard APIs
- version control (because lots of versions will be generated)

RAD USES "TIMEBOXING"

- Secondary features are dropped as necessary to stay on schedule.

RAD USES ITERATIVE, EVOLUTIONARY PROTOTYPING

- JAD (Joint Application Development) MEETING High-level end-users and designers meet in a brainstorming session to generate a rough list of initial requirements.
    - Developers talk and listen
    - Customers talk and listen
- ITERATE UNTIL DONE
    - Developers build / evolve prototype based on current requirements.
    - Designers review the prototype.
    - Customers try out the prototype, evolve their requirements.
    - FOCUS GROUP meeting
      Customers and developers meet to review product together, refine requirements, generate change requests.
        - Developers listen.
        - Customers talk.
    - Requirements and change requests are "time boxed".
        - Changes that cannot be accommodated within existing time boxes are eliminated.
        - If necessary to stay in the box, requirements are dropped.

There are several advantages to using the rapid application development methodology. "The primary advantage of RAD is obvious; information systems developed in as little as one quarter the usual time. Shorter development cycles also mean cheaper systems, as fewer organizational resources need to be devoted to develop any particular system" (Hoffer et al. 1999). Also, because RAD uses smaller development teams, there is an additional cost savings (Martin 1991). Lastly, because the system is developed in such a short period of time, it will more closely match the current business needs and therefore be of more value (Hoffer et al. 1999).

Conversely, there can be several disadvantages to using RAD. "For some, RAD will always stand for 'Rough and Dirty' development— an excuse for sidetracking the disciplines of software engineering standards" (Howard 2002). "RAD does have drawbacks: with its emphasis on developing systems quickly, the detailed business models that underlie information systems are often neglected, leading to the risk that systems may be out of alignment with the overall business. Similarly, the speed of development may lead to analysts overlooking systems engineering concepts such as consistency, programming standards, module reuse, scalability, and systems administration" (Hoffer et al. 1999). Table 2.2 gives a nice synopsis of the advantages and disadvantages of RAD (Hoffer 1999).

Table 2.2 Advantages and Disadvantages of Rapid Application Development (Hoffer 1999)

Advantages	*Disadvantages*
Dramatic time savings during the system development effort.	More speed and lower cost may lead to lower overall system quality (due to lack of controls)
Can save time, money, and human effort	Danger of misalignment of system developed via RAD with the business due to missing information on underlying business processes.
Tighter fit between user requirements ad system specifications.	May have inconsistent internal designs within and across systems.
Works especially well where speed of development is important, as with rapidly changing business conditions or where systems can capitalize on strategic opportunities.	Possible violation of programming standards related to inconsistent naming conventions and insufficient documentation.
Ability to rapidly change system design as demanded by users	Difficulties with module reuse for future systems
System optimized for users involved in RAD process	Lack of scalability designed into system
Concentrates on essential system elements from user viewpoint.	Lack of attention to later systems administration built into system
Strong user stake and ownership of system	High cost of commitment on the part of key user personnel

2.7 Object Oriented Development

Since the late 1980's, object-oriented design has been widely publicized and adopted (Sommerville 1996). While conventional systems development is based on decomposing a system into procedures (process oriented) or data (data oriented), Object Oriented Systems Development is predicated on decomposing a problem into interacting objects that encapsulate both data and behavior (Booch 1994). An object's behavior (functions, procedures, or operations) is actuated when it receives messages (function calls) from other objects. The object sending the message need only know what is to be done, not how to do it. The object receiving the message contains the implementation details. The encapsulation of data and behavior into a single entity (i.e., the object) is expected to provide many benefits over conventional methods (Hargrave 1997).

It should be noted that object-oriented development, which is the focus of this paper, is made up of several sub-categories. Those include: (Coad/Yourdon 1991, Sommervile 1996).

- Object-oriented analysis is concerned with developing an object oriented model of the application domain.
- Object-oriented design is concerned with developing an object-oriented model of a software system to implement the identified requirements.
- Object-oriented programming is concerned with realizing a software design using an object-oriented programming language

All phases of the OOSD belong to the OO paradigm (Korson and McGregor 1990 as reported by Johnson 1999). The OO paradigm can be explained in terms of five basic concepts (Johnson 1999). Those concepts are:

- Object – an encapsulation of information and the description of its manipulation (Coad/Yourdon 1991).

- Class – a mechanism for classifying variables and methods according to their similarities and special case extensions (Coad/Yourdon 1991).

- Inheritance – a mechanism that simplifies the definition of software components that are similar to those previously defined (Coad/Yourdon 1991).

- Polymorphism – an important object-oriented programming concept in which objects from two or more different classes respond to the same set of messages (Becker, Rasala, Bergin, Shannon, and Wallingford 2001).

- Dynamic binding – A mechanism by which, when the compiler can't determine which method implementation to use in advance, the system selects the appropriate method at runtime, based on the class of the object.

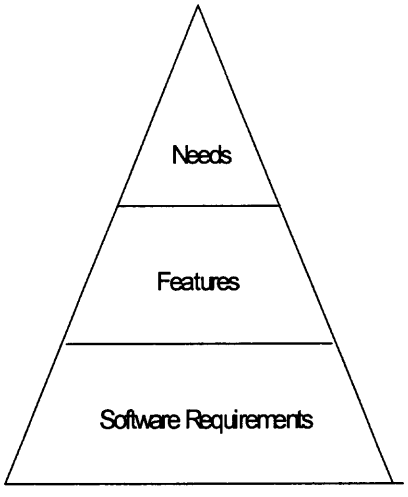


Figure 2.5 The Object oriented process.
Source: Leffingwell 2001

The OO process can generally be thought of as an iterative process whereby requirements are defined and then solved in increasing levels of detail. Some have defined this process as being synonymous with peeling an onion (Hoffer 1999) or to a pyramid of increased specificity as defined by Leffingwell (2001).

In Leffingwell's notation, the problem domain is defined in terms of stakeholders' needs. The solution domain is then generally defined in terms of functions which are services that the system provides to fulfill one or more stakeholder needs. Features are represented in natural language, using terms familiar to the user. Features are then broken down into software requirements. At this level, specify requirements and use cases sufficiently for developers to write code and testers to see whether the code meets the requirements. Leffingwell (2001)

Leffingwell then further dissects software requirements into three categories. They are functional requirements, nonfunctional requirements, and design constraints. Functional requirements express what the system does. Nonfunctional requirements focus on specifying additional system "attributes," such as performance requirements, throughput, usability, reliability, and supportability. Leffingwell defines a design constraint as a restriction upon the design of a system, or the process by which a system is developed, that does not affect the external behavior of the system, but must be fulfilled to meet technical, business, or contractual obligations (Leffingwell 2001).

There are many OO design methods:

- Coad and Yourdon 1990
- Robinson 1992
- Jacobson, Booch, and Rumbaugh 1999
- Booch 1994
- Graham 1994

These methods all have the following in common: (Sommerville 1996)

- The identification of the objects in the system along with their attributes and operations

- The organization of objects into an aggregation hierarchy which shows how objects are a 'part-of' other objects
- construction of dynamic 'object-use' diagrams that show which object services are used by other objects
- The specification of object interfaces

The Unified Modeling Language or UML has become the industry standard for OO development. The development of UML began in late 1994 when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. In the fall of 1995, Ivar Jacobson and his Objectory Company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method. (http://cgi.omg.org/news/pr97/umlprimer.html) Eventually this process evolved to become the Rational Unified Process or RUP (http://www.rational.com). Figure 2.6 shows the evolution of the process to its current state.

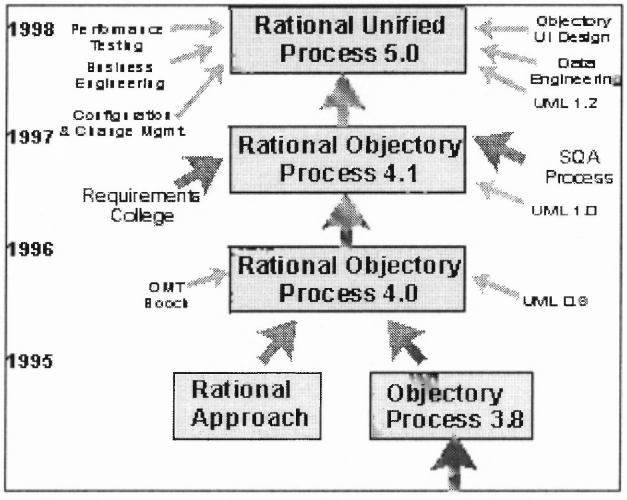


Figure 2.6 The evolution of the Rational Unified Process.
Source: http://www.rational.com

"RUP is a Software Engineering Process. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end-users, within a predictable schedule and budget." (Krutchen 2000)

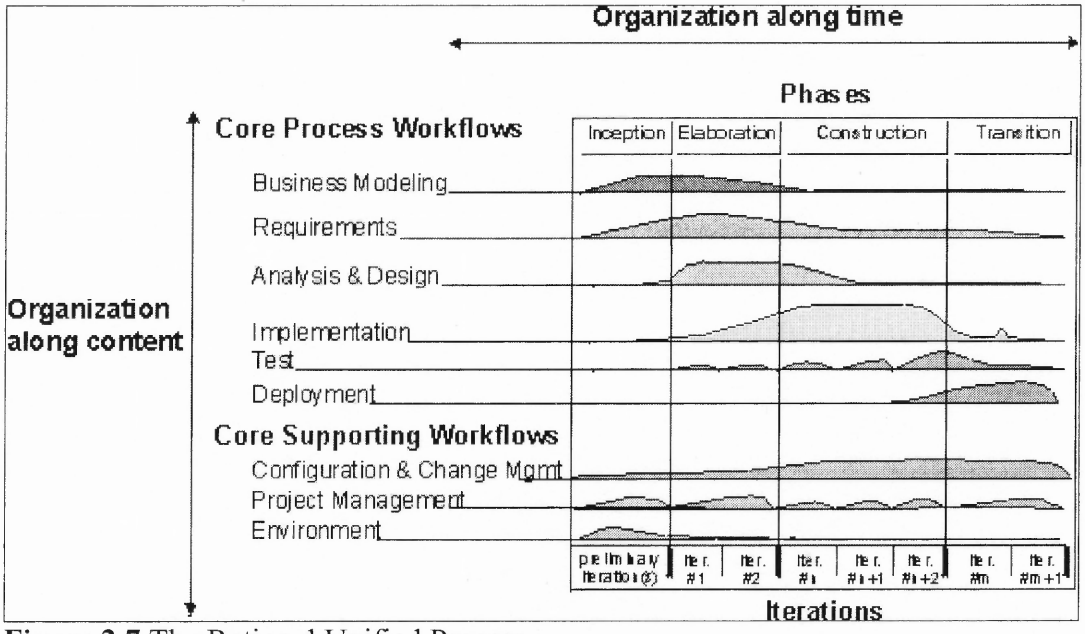


Figure 2.7 The Rational Unified Process.
Source: http://www.rational.com

The RUP process, shown in Figure 2.7, can be described in two dimensions or along two axes (http://www.rational.com). The horizontal axis shows the organization of the project in terms of time and the vertical axis shows the areas of workflows or disciplines in a software development project. The height of the bar associated with each discipline is larger or smaller based on the time that is spent on that discipline during a phase (Hirsch 2002).

There are nine core workflows in RUP (Kruchten 2000):

- Business modeling – document business processes using business use cases
- Requirements – elicit, organize, and document required functionality and constraints. Create a vision document.
- Analysis & Design – show how the system will be realized in the implementation phase.
- Implementation – Writing and debugging source code, unit testing, and build management.
- Test – Integration-, system- and acceptance testing.
- Deployment – successfully produce product releases, and deliver the software to its end users
- Project Management – the art of balancing competing objectives, managing risk, and overcoming constraints to deliver, successfully, a product which meets the needs of both customers and the users
- Configuration and Change Management workflow – describe how to control the numerous artifacts produced by the many people who work on a common project.
- Environment workflow – provide the software development organization with the software development environment—both processes and tools—that are needed to support the development team.

There are four phases in RUP (Kruchten 2000). Within each phase there may be several iterations that are performed until a milestone is met. Each iteration builds on the results of the previous iteration and delivers an executable release of the system. The duration and goals of an iteration are planned before an iteration starts. When an iteration is completed, a full assessment of the iteration is done in order to allow for corrective action if needed (Hirsch 2002).

The four phases are (Kruchten 2000):

- Inception phase – establish the business case for the system and delimit the project scope.
- Elaboration phase – analyze the problem domain, establish a sound architectural foundation, develop the project plan, and eliminate the highest risk elements of the project.
- Construction phase – components and application features are developed and integrated into the product, and all features are thoroughly tested.
- Transition phase – transition the software product to the user community.

No explanation of OO development and RUP would be complete without a discussion of UML. At the heart of RUP is the Unified Modeling Language (UML). UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. UML uses mostly graphical notations to express the design of software projects. Using UML helps project teams communicate, explore potential designs, and validate the architectural design of the software (http://cgi.omg.org/news/pr97/umlprimer.html).

There are several diagrams developed in UML. Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction. Figure 2.8 explains several common UML diagrams.

Use Case Diagram – displays the relationship among actors and use cases.

Class Diagram - models class structure and contents using design elements such as classes, packages and objects. It also displays relationships such as containment, inheritance, associations and others.

Interaction Diagrams:

Sequence Diagram - displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects).

Collaboration Diagram - displays an interaction organized around the objects and their links to one another. Numbers are used to show the sequence of messages.

State Diagram displays the sequences of states that an object of an interaction goes through during its life in response to received stimuli, together with its responses and actions.

Activity Diagram displays a special state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. This diagram focuses on flows driven by internal processing.

Physical Diagrams:

Component Diagram displays the high level packaged structure of the code itself. Dependencies among components are shown, including source code components, binary code components, and executable components. Some components exist at compile time, at link time, at run times well as at more than one time.

Deployment Diagram displays the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units.

Figure 2.8 Uml Diagrams.
Source: http://cgi.omg.org/news/pr97/umlprimer.html

There are many advantages to the OO paradigm. These advantages include (Johnson 1999):

- An easier modeling process.
- Better analysis and design models.
- Easier transition from analysis to design to implementation.
- Improved communication between developers and users.
- Improved communication among developers.
- Easier, more flexible development using software components.
- Decreased development time.
- Higher system quality.
- More effective forms of modularity.
- Increased reuse of analysis and design models.
- Increased reuse of program code.
- More stable system designs.
- Less system maintenance.
- Easier system maintenance.

The most commonly cited disadvantages of using OOSD are (Johnson 1999):

- Increased development time.
- Poorer run-time performance.

2.8 The Light/Agile Methods

A recent study by the Cutter Consortium found that traditional SDLC methodologies "fall short in the new e-business environment. They are unable to keep up with fast-paced, ever-changing e-business projects" (Cutter 2000). "...a new group of methodologies have appeared in the last few years. For a while these were known as lightweight methodologies, but now the accepted term is agile methodologies. For many people the appeal of these agile methodologies is their reaction to the bureaucracy of the monumental (heavy) methodologies. These new methods attempt a useful compromise between no process and too much process, providing just enough process to gain a reasonable payoff" (Fowler website).

These new agile methodologies differ from the heavy methodologies on three main issues (Fowler):

- ***Agile methods are code-oriented rather than document oriented.*** That means that there is less formal documentation and that there is an assumption that the source code generated is in itself documentation.

- ***Agile methods are adaptive rather than prescriptive.*** Heavy methods tend to try to plan out a large part of the software process in great detail for a long span of time, this works well until things change. So their nature is to resist change. The agile methods, however, welcome change. They try to be processes that adapt and thrive on change, even to the point of changing themselves.

- ***Agile methods are people-oriented rather than process-oriented.*** They explicitly make a point of trying to work with peoples' nature rather than against them and to emphasize that software development should be an enjoyable activity.

The agile movement in software development began in 2001 when a group of consultants and practitioners got together and created the "Manifesto for Agile Software Development" (Beck et al. 2001). The manifesto is as follows (http://www.agilemanifesto.org/):

Manifesto for Agile Software Development

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more."

This group also published a list of principles to which agile software development should adhere (http://www.agilemanifesto.org/):

- The highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

There have been several methods introduced under the "agile" umbrella in the last few years. The proponents of these methods describe many successful implementations by practitioners in applying these methods to real-world situations (Beck 2000, Cockburn2002, Schwaber 2002). The biggest criticism of these methods from Academia has been the lack of empirical evidence supporting the claims of their benefits and their lack of theoretical foundation (Abrahamsson et al. 2003).

However, proponents would argue that there are theoretical foundations for at least some of the agile methods such as Scrum (Schwaber 2002) and the Adaptive Method (Highsmith 2000). Furthermore, there are striking similarities between the tenets of the agile philosophy and the idea of "A-methodological Systems Development" (Truex et al. 2000). Truex et al., make an excellent argument for using "a-methods" of software development in certain situations. Those arguments are based on numerous empirical studies.

In the next several sections, the most popular of these new agile methodologies will be described.

2.9 Extreme Programming

The most prominent of the new agile methodologies is Extreme Programming (XP) (Booch 2001). "What is XP? XP is a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop software" (Beck 2000). It is distinguished from other methodologies by (Beck 2000):

- Its early, concrete, and continuing feedback from short cycles.
- Its incremental planning approach, which quickly comes up with an overall plan that is expected to evolve through the life of the project.
- Its ability to flexibly schedule the implementation of functionality, responding to changing business needs.
- Its reliance on automated tests written by programmers and customers to monitor the progress of development, to allow the system to evolve, and to catch defects early.
- It reliance on oral communication, tests, and source code to communicate system structure and intent.
- Its reliance on an evolutionary design process that lasts as long as the system lasts.
- Its reliance on the close collaboration of programmers with ordinary skills.
- Its reliance on practices that work with both the short-term instincts or programmers and the long-term interests of the project.

XP is designed to be used primarily by small to medium sized development teams with two to ten programmers (Beck 2000). Its process is comprised of 12 practices (http://www.xprogramming.com):

- ***The Planning Game*** – The XP planning process allows the XP "customer" to define the business value of desired features, and uses cost estimates provided by the programmers, to choose what needs to be done and what needs to be deferred. The effect of XP's planning process is that it is easy to steer the project to success.

- ***Small Releases*** – XP teams put a simple system into production early, and updates it frequently on a very short cycle.

- ***Metaphor*** – XP teams use a common "system of names" and a common system description that guides development and communication.

- ***Simple Design*** – A program built with XP should be the simplest program that meets the current requirements. There is not much building "for the future". Instead, the focus is on providing business value. Of course it is necessary to ensure that you have a good design, and in XP this is brought about through "re-factoring", discussed below.

- ***Testing*** – XP teams focus on validation of the software at all times. Programmers develop software by writing tests first, then software that fulfills the requirements reflected in the tests. Customers provide acceptance tests that enable them to be certain that the features they need are provided.

- ***Re-factoring*** – XP teams improve the design of the system throughout the entire development. This is done by keeping the software clean: without duplication, with high communication, simple, yet complete.

- ***Pair Programming*** – XP programmers write all production code in pairs, two programmers working together at one machine. Pair programming has been shown by many experiments to produce better software at similar or lower cost than programmers working alone.

- ***Collective Ownership*** – All the code belongs to all the programmers. This lets the team go at full speed, because when something needs changing, it can be changed without delay.

- ***Continuous Integration*** – XP teams integrate and build the software system multiple times per day. This keeps all the programmers on the same page, and enables very rapid progress. Perhaps surprisingly, integrating more frequently tends to eliminate integration problems that plague teams who integrate less often.

- ***40-hour Week*** – Tired programmers make more mistakes. XP teams do not work excessive overtime, keeping them fresh, healthy, and effective.

- ***On-site Customer*** – An XP project is steered by a dedicated individual who is empowered to determine requirements, set priorities, and answer questions as the programmers have them. The effect of being there is that communication improves, with less hard-copy documentation – often one of the most expensive parts of a software project.

- ***Coding Standard*** – For a team to work effectively in pairs, and to share ownership of all the code, all the programmers need to write the code in the same way, with rules that make sure the code communicates clearly.

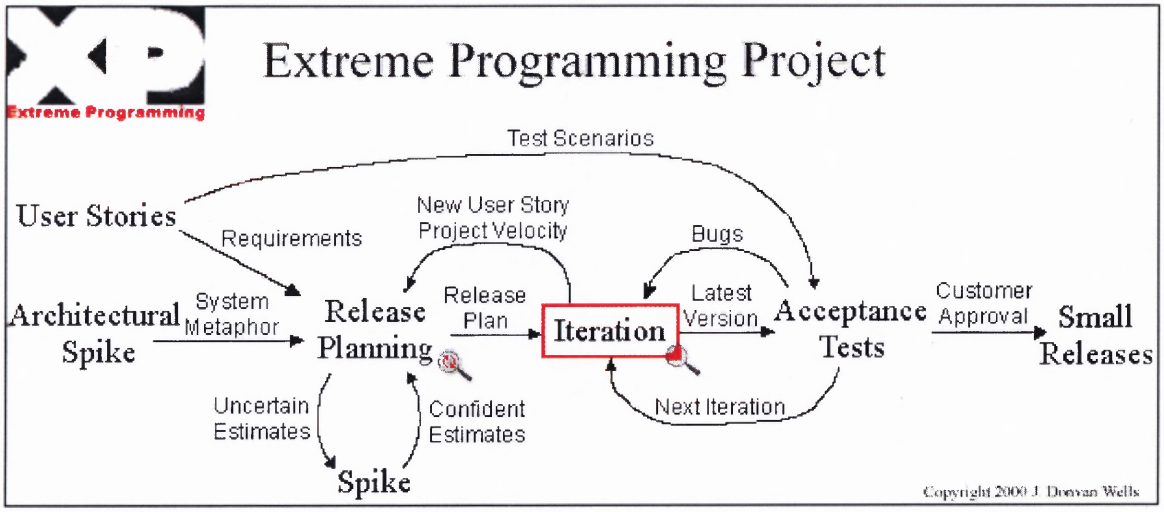


Figure 2.9 Extreme programming.
Source: http://www.extremeprogramming.org/

Figure 2.9 shows a typical flowchart for an XP project. This diagram demonstrates how XP uses user stories and system metaphor to build a release plan. The software is then developed using an iterative cycle of programming, testing and debugging. Finally customer approval is given and that small release is accepted.

Watts Humphrey lists the advantages of XP

(http://www.computer.org/Seweb/Dynabook/HumphreyCom.htm 2002):

- Emphasis on customer involvement: A major help to projects where it can be applied.
- Emphasis on teamwork and communication: As with the traditional methodology, this is very important in improving the performance of just about every software team.
- Programmer estimates before committing to a schedule: This helps to establish rational plans and schedules and to get the programmers personally committed to their schedules-a major advantage of XP and traditional approaches.
- Emphasis on responsibility for quality: Unless programmers strive to produce quality products, they probably won't.
- Continuous measurement: Since software development is a people-intensive process, the principal measures concern people. It is therefore important to involve the programmers in measuring their own work.
- Incremental development: Consistent with most modern development methods.
- Simple design: Though obvious, worth stressing at every opportunity.
- Frequent redesign, or re-factoring: A good idea but could be troublesome with any but the smallest projects.
- Having engineers manage functional content: Should help control function creep.
- Frequent, extensive testing: Cannot be overemphasized.
- Continuous reviews: A very important practice that can greatly improve any programming team's performance (few programmers do reviews at all, let alone continuous reviews).

In recent years there is a growing body of literature being written on what is wrong with XP (Choi and Deek 2002). However Humphrey seems to sum it up best with his list of disadvantages

(http://www.computer.org/Seweb/Dynabook/HumphreyCom.htm 2002):

- Code-centered rather than design-centered development: Although the lack of XP design practices might not be serious for small programs, it can be disastrous when programs are larger than a few thousand lines of code or when the work involves more than a few people.

- Lack of design documentation: Limits XP to small programs and makes it difficult to take advantage of reuse opportunities.

- Producing readable code (XP's way to document a design) has been a largely unmet objective for the last 40-plus years. Furthermore, using source code to document large systems is impractical because the listings often contain thousands of pages.

- Lack of a structured review process: When engineers review their programs on the screen, they find about 10-25% of the defects. Even with pair programming, unstructured online reviews would still yield only 20-40%. With the traditional structured review process, most engineers achieve personal review yields of 60-80%, resulting in high-quality programs and sharply reducing test time.

- Quality through testing: A development process that relies heavily on testing is unlikely to produce quality products. The lack of an orderly design process and the use of unstructured reviews mean that extensive and time-consuming testing would still be needed, at least for any but the smallest programs.

- Lack of a quality plan: We have found with the traditional process that quality planning helps properly trained teams produce high-quality products, and it reduces test time by as much as 90%. XP does not explicitly plan, measure, or manage program quality.

- Data gathering and use: We have found with the traditional process that, unless the data are precisely defined, consistently gathered, and regularly checked, they will not be accurate or useful. The XP method provides essentially no data-gathering guidance.

- Limited to a narrow segment of software work: Since many projects start as small efforts and then grow far beyond their original scope, XP's applicability to small

teams and only certain kinds of management and customer environments could be a serious problem.

- Methods are only briefly described: While some programmers are willing to work out process details for themselves, most engineers will not. Thus, when engineering methods are only generally described, practitioners will usually adopt the parts they like and ignore the rest. Kent Beck notes that, when the XP method fails in practice, this is usually the cause.

- Obtaining management support: The biggest single problem in introducing any new software method is obtaining management support. The XP calls for a family of new management methods but does not provide the management training and guidance needed for these methods to be accepted and effectively practiced.

- Lack of transition support: Transitioning any new process or method into general use is a large and challenging task. Successful transition of any technology requires considerable resources, a long-term support program, and a measurement and analysis effort to gather and report results. XP provides no such support.

Perhaps the two most controversial traits of XP are the facts that it produces little documentation and it is usually limited to small teams (and therefore small projects) (Cockburn 2002). Allistair Cockburn explains that "the planning game" can be used to explicitly call for documentation on systems where it is perceived that additional documentation may be needed. Also, Cockburn defends the team size limitations of XP, by stating that a smaller team utilizing XP can accomplish the same goals as a larger team using a different methodology. He does, however, admit that XP has its limits (Cockburn 2002).

2.10 Crystal Family

"Crystal is a family of human-powered and adaptive, ultra-light, "shrink-to-fit" software development methodologies. "Human-powered" means that the focus is on achieving project success through enhancing the work of the people involved (other methodologies might be process-centric, or architecture-centric, or tool-centric, but Crystal is people-centric). "Ultra-light" means that for whatever the project size and priorities, a Crystal-family methodology for the project will work to reduce the paperwork, overhead and bureaucracy to the least that is practical for the parameters of that project. "Shrink-to-fit" means that you start with something possibly small enough, and work to make it smaller and better fitting. Crystal is non-jealous, meaning that a Crystal methodology permits substitution of similar elements from other methodologies"

(http://crystalmethodologies.org).

The Crystal Methodologies are in actuality a "kit" of methodologies, whereby the correct tool or method can be used based on the circumstances of the project. The two factors affecting what method should be used are the number of people working on the project and the criticality of the system being developed.

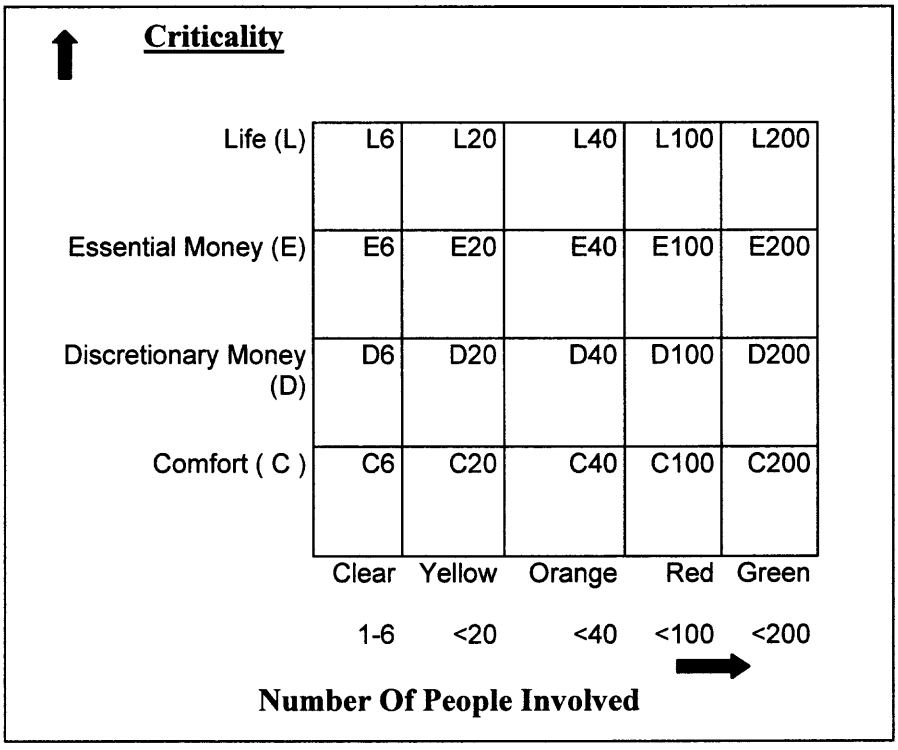


Figure 2.10 Crystal methodologies.
Source: Cockburn 2002

Figure 2.10 shows a matrix that can be used to select the proper Crystal tool by using the number of people and the criticality of the system as parameters. The shading of each cell becomes darker as you move higher and further to the right. The bottom left cell is clear because it involves the least number of people and has the least critical system. The upper right cell would be the darkest as it involves the largest number of people and the most critical system (Cockburn 2002).

There are several values, principles, and rules to the Crystal methodology. There is also a core philosophy that software development is viewed as a cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game (Cockburn 2002).

The two values of Crystal Methodologies are that they are:

- People and communication-centric- meaning that "tools, work products, and processes are there only to support the human component" (Cockburn 2002).
- Highly tolerant – meaning that it recognizes varying human cultures.

The main principles of Crystal are: (http://crystalmethodologies.org)

- Every project needs a slightly different set of policies and conventions, or methodology.
- The workings of the project are sensitive to people issues, and improve as the people issues improve, individuals get better, and their teamwork gets better.
- Better communications and frequent deliveries communication reduce the need for intermediate work products

The two rules common to the Crystal family are (Cockburn 2002):

- The project must use incremental development, with increments of four months or less (and a strong preference for one-to three month increments).
- The team must hold pre- and post-increment reelection workshops (with a strong preference for holding mid-increment reflection workshops are well).

The two base techniques in Crystal are (Cockburn 2002):

- The methodology tuning technique: using project interviews and a team workshop to convert a base methodology to a starter methodology for the project.
- The technique used to hold the reflection workshop.

The primary advantage of the Crystal Methodologies over some of the other agile methods (XP for instance) is that the Crystal methods choose the best course of action based upon the current cultural environment. "Crystal and Adaptive methodologies say, 'what is the cultural context, and within that context, how can we set up the project to deal with shifting requirements... and by the way, how can we set it up to deal with cultural shifts, too.' (where 'cultural' is to be liberally interpreted as size, criticality,

geographic dispersion, and all that "people stuff") (http://crystalmethodologies.org). This philosophy of the Crystal methodologies makes them more able to adapt to different situations, where some of the methods such as XP are limited in their applicability.

Conversely, the main disadvantage to Crystal is that because it is "shrink to fit" it must be constantly fine-tuned to the project, whereby XP is more of a "canned" approach.

2.11 Scrum

"Scrum is an agile, lightweight process that can be used to manage and control software and product development. Wrapping existing engineering practices, including Extreme Programming, Scrum generates the benefits of agile development with the advantages of a simple implementation. Scrum significantly increases productivity while facilitating adaptive, empirical systems development" (http://www.controlchaos.com).

Scrum is an iterative, incremental process for developing any product or managing any work. It produces a potentially shippable set of functionality at the end of every iteration. Its attributes are (http://www.controlchaos.com):

- Scrum is an agile process to manage and control development work.
- Scrum is a wrapper for existing engineering practices.
- Scrum is a team-based approach to iteratively, incrementally develop systems and products when requirements are rapidly changing.
- Scrum is a process that controls the chaos of conflicting interests and needs.
- Scrum is a way to improve communications and maximize co-operation.
- Scrum is a way to detect and cause the removal of anything that gets in the way of developing and delivering products.
- Scrum is a way to maximize productivity.
- Scrum is scalable from single projects to entire organizations. Scrum has controlled and organized development and implementation for multiple interrelated products and projects with over a thousand developers and implementers.
- Scrum is a way for everyone to feel good about their job, their contributions, and that they have done the very best they possibly could.
- Scrum is a pattern.

Figure 2.11 shows how the Scrum process works. Scrum divides a project into iterations which last 30 days. Each of these iterations is called a "sprint". Before you

begin a sprint you define the functionality required for that sprint and then leave the team to deliver it. During the sprint, the requirements are not allowed to change (Schwaber 2001).

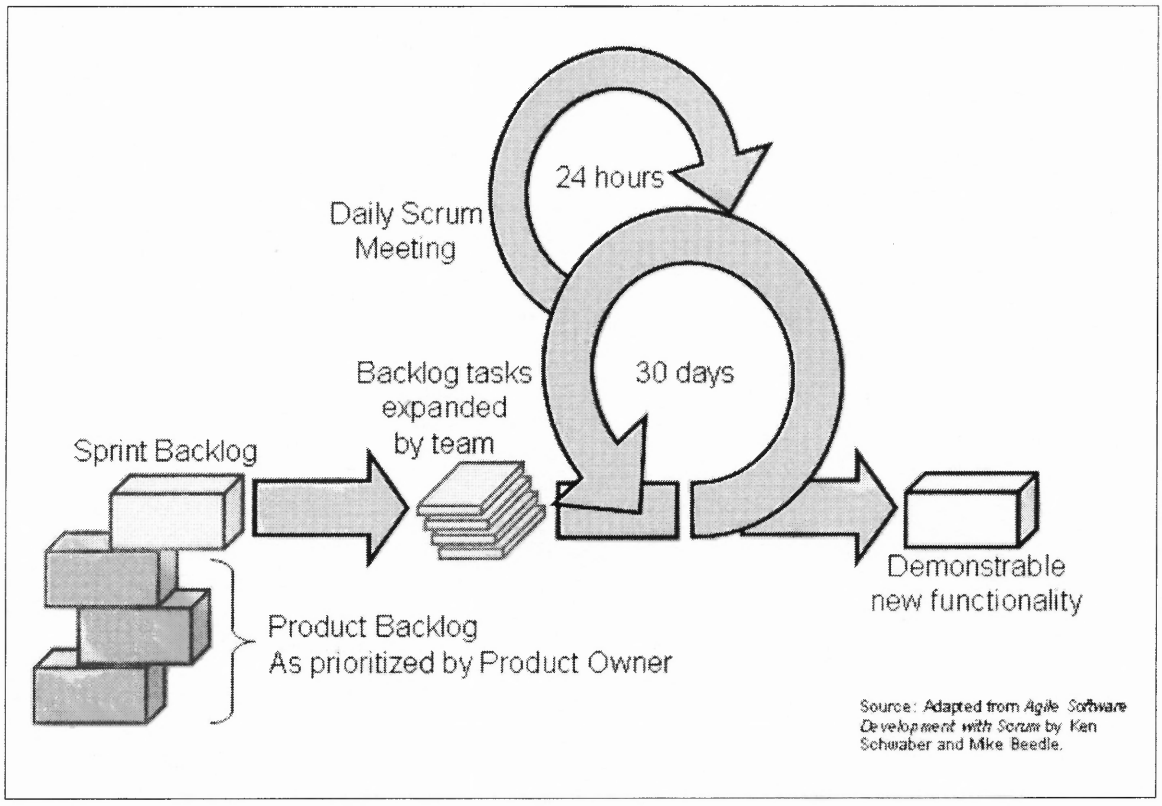


Figure 2.11 Scrum.
Source: Schwaber 2001

The team has a meeting every 24 hours called a "scrum", which is based on the huddle used in a rugby match. These are short 15 minute meetings, where the team discusses any problems it has encountered since the last scrum and/or what it will accomplish before the next scrum. This allows management to get a daily briefing as well as provide answers to any problems that have come up since the day before (Fowler 2002). These daily scrums also allow the team to fine tune the activities of the sprint to any changes in the environment.

All requests for changes or additions to the product functionality are added to the product backlog. One person (the product owner) prioritizes all requests submitted. In preparation for a sprint, the higher priority items are chosen to be implemented first. Even though all requests are added to the backlog, lower priority items (or those deemed not worthy of doing) may never get done.

At the end of a sprint there is a control called a "sprint review", where the accomplishments of the sprint are reviewed and inspected. Management, customers, and the team inspect the product and then decide what to do next. This could include re-working the team or bringing in new tools for the next sprint (Schwaber 2001).

At the heart of the Scrum technique is the idea of the "empirical model of process control". It provides and exercises control through frequent inspection and adaptation for processes that are imperfectly defined and generate unpredictable and unrepeatable outputs (Schwaber 2001). This is in direct contrast to the heavy system development techniques which employ a "defined" process control model. This model requires a well defined set of inputs and produces the same set of outputs each time.

The two main criticisms of Scrum are that it does not provide an adequate process for estimating resources up front, and that it is limited to small teams. The creators of Scrum address the first problem by arguing that it is a myth that requirements can be adequately stated at the beginning of a project. Scrum estimates resources needed for a sprint at a time. After a few sprint iterations, management is able to get the idea of what the project will cost. The problem of the limitation to small teams, is addressed by allowing multiple small teams to work on a project at the same time. Each team is involved in their own sprint which can occur simultaneously.

2.12 Feature Driven Development

"Like all good software development processes, Feature Driven Development is built around a core set of 'best practices'. The chosen practices are not new but this particular blend of the ingredients is new. Each practice compliments and reinforces the others. The result is a whole greater than the sum of its parts; there is no single practice that underpins the whole process. A team could choose to implement just one or two of the practices, but would not get the full benefit that occurs by using the whole FDD process...." (Palmer 2002). FDD is a model-driven short-iteration process. It begins with establishing an overall model shape. Then it continues with a series of two-week "design by feature, build by feature" iterations (Coad 1999).

FDD starts with the creation of a domain object model in collaboration with domain experts. Using information from the modeling activity and from any other requirements activities that have taken place, the developers go on to create a features list. Next a rough plan, is drawn up and responsibilities are assigned. Then small dynamically-formed teams develop the features by repeatedly performing design and build iterations that last no longer than two weeks and are often much shorter (Palmer 2002).

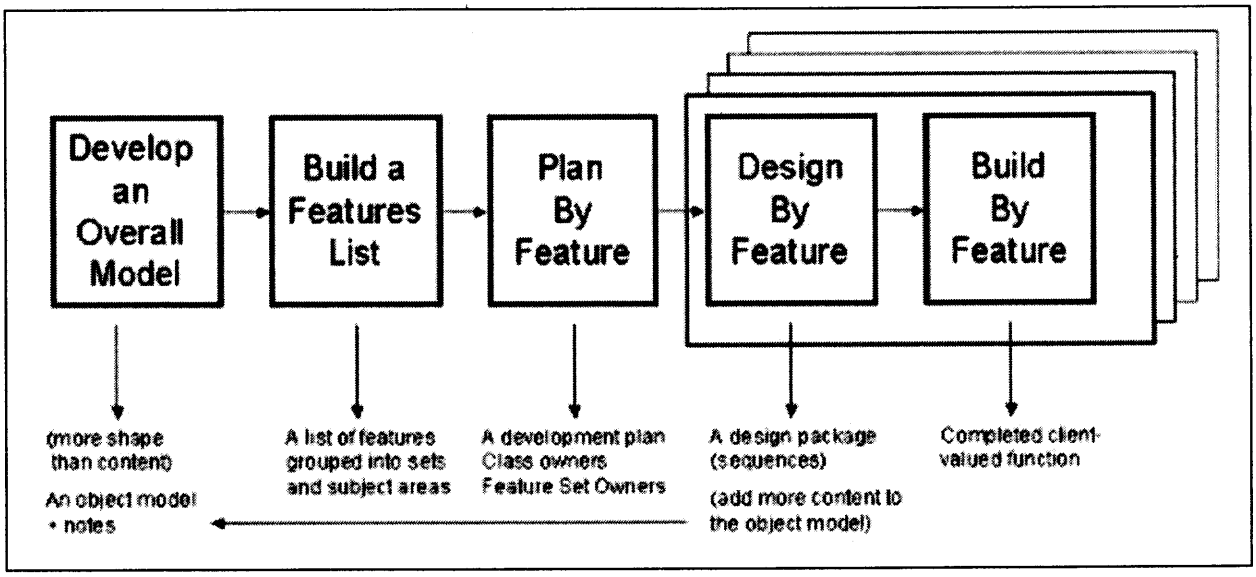


Figure 2.12 Feature driven development.
Source: Palmer 2002

Figure 2.12 illustrates the five processes of FDD. These processes are (Coad 1999):

- Develop an overall model (using initial requirements/features, snap together with components, focusing on shape).
- Build a detailed, prioritized features list.
- Plan by feature.
- Design by feature (using components, focusing on sequences).
- Build by feature.

The processes are described by Stephen Palmer as follows:

- **Develop an overall model** – Domain Experts perform a high-level walkthrough of the scope of the system and its context. They then perform detailed domain walkthroughs for each area of the domain that is to be modeled. After each domain walkthrough, small groups are formed with a mix of domain and development staff. Each small group composes its own model in support of the domain walkthrough and presents its results for peer review and discussion. One of the proposed models or a merge of the models is selected by consensus and becomes the model for that domain area. The domain area model is merged into the overall model, adjusting model shape as required (Palmer 2002).

- **Build a detailed, prioritized features list** – A team usually comprising just the chief programmers from process 1 is formed to decompose the domain functionally. Based on the partitioning of the domain by the domain experts in process 1, the team breaks the domain into a number of areas (major feature sets). Each area is further broken into a number of activities (feature sets). Each step within an activity is identified as a feature. The result is a hierarchically categorized features list (Palmer 2002).

- **Plan by feature** – The Project Manager, Development Manager, and Chief Programmers plan the order that the features are to be implemented, based on feature dependencies, load across the development team, and the complexity of the features to be implemented. The main tasks in this process are not a strict sequence. Like many planning activities, they are considered together, with refinements made from one or more tasks, then considering the others again. A typical scenario is to consider the development sequence, then consider the assignment of feature sets to Chief Programmers and, in doing so, consider which of the key classes (only) are assigned to which developers (remembering that a Chief Programmer is also a developer). When this balance is achieved and the development sequence and assignment of business activities to Chief Programmers is essentially completed, the class ownership is completed (beyond the key classes that were already considered for ownership) (Palmer 2002).

- **Design by feature** – A number of features are scheduled for development by assigning them to a Chief Programmer. The chief programmer selects features for development from his or her "inbox" of assigned features. Operationally, it is often the case that the Chief Programmer schedules small groups of features at a time for development. He or she may choose multiple features that happen to use the same classes (therefore, developers). Such a group of features forms a *chief programmer work package*. The Chief Programmer then forms a feature team by identifying the owners of the classes (developers) likely to be involved in the development of the selected feature(s). This team produces the detailed sequence diagram(s) for the selected feature(s). The Chief Programmer then refines the object model, based on the

content of the sequence diagram(s). The developers write class and method prologues. A design inspection is held (Palmer 2002).

- **Build by feature** – Working from the design package produced during the Design by Feature process, the Class Owners implement the items necessary for their classes to support the design for the feature(s) in the work package. The code developed is then unit-tested and code-inspected, the order of which is determined by the Chief Programmer. After a successful code inspection, the code is promoted to the build (Palmer 2002).

FDD is promoted as combining the best of both the heavy and agile worlds (Coad 1999, Palmer 2002). As we have seen, the two main problems of all of the agile methods written about in this paper so far have been lack of an overall plan and scalability. FDD addresses the problem of planning by first creating the overall model. This model allows all stakeholders to get a good initial picture of what the system is supposed to do. However, because it is "more shape than content", developers do not get bogged down in developing this model nor are they required to adhere to it as they progress through the development iterations.

The problem of scalability to larger projects is addressed by the nature of the FDD process. Instead of ALL team members attending ALL meetings (as in XP and Scrum), only selected members are required to attend the higher level meetings. There is in essence a hierarchical separation of duties with members at the higher end of the scale focusing on general tasks and those at the lower focusing on specific.

One disadvantage to the Feature Drive Approach is increased complexity. This is because often times the individual features are not documented explicitly but rather, implicitly through code. Also, this layered approach can hinder the application of multidimensional features (Greefhorst 2000).

2.13 Dynamic System Development

"In a nutshell, the Dynamic System Development Method (DSDM) is a game-changing, non-proprietary agile application development project model for developing business solutions within tight timeframes. It shortens the clock-speed (and time to market) for delivery of core business benefits. DSDM is the only approach that can guarantee delivery on an exact day under tight, Internet-time deadlines. It's tool-independent – there are no tools or software packages to buy (or be hamstrung by)." (http://www.surgeworks.com)

DSDM is made up of seven phases; Pre-Project, Feasibility Study, Business Study, Functional Model Iteration, Design and Build Iteration, Implementation, and Post-Project. The Pre-Project, Feasibility and Business Studies are done sequentially at the start of the project. The rest of the development phases are iterative and incremental and are coordinated based on the demands of the individual project. The Post-Project phase checks to make sure that the expected business benefits have been achieved.

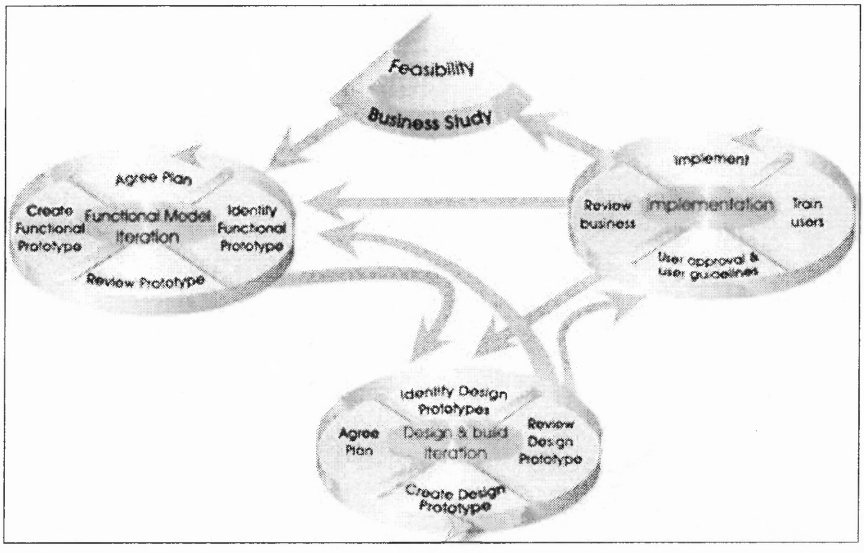


Figure 2.13 Dynamic system development method.
Source: http://www.dsdm.org

Figure 2.13 illustrates the DSDM process. The phases can be explained as follows (http://www.dsdm.org):

- **The Pre-Project Phase** – ensures that only the right projects are started and that they are set up correctly. Once it has been determined that a project is to go ahead, funding is available, etc., the initial project planning for the Feasibility Study is done.

- **Feasibility Study** – determines whether DSDM is the right approach for the project. Defines the problem to be addressed and assesses the likely costs and the technical feasibility of delivering a system to solve the business problem. The Feasibility Study should not last more than a few weeks.

- **Business Study** – Studies the business processes affected and their information needs. The short timescales of a DSDM project mean that this activity has to be very strongly collaborative, using a series of facilitated workshops attended by knowledgeable and empowered staff that can quickly pool their knowledge and gain consensus as to the priorities of the development. The result of these workshops will be the Business Area Definition which will not only identify the business processes and associated information but also the classes (or types) of users who will be affected in any way by the introduction of the system.

- **Functional Model Iteration** – Refines the business-based aspects of the system, i.e., building on the high-level processing and information requirements identified during the Business Study. Both the Functional Model Iteration and the Design and Build Iteration consist of cycles of four activities:

  - Identify what is to be produced.
  - Agree how and when to do it.
  - Create the product.
  - Check that it has been produced correctly (by reviewing documents, demonstrating a prototype or testing part of the system).

  The bulk of development work is in the two iteration phases where prototypes are incrementally built towards the tested system. All prototypes in DSDM are intended to evolve into the final system and are therefore built to be robust enough for operational use and to satisfy any relevant non-functional requirements, such as performance.

- **Design and Build Iteration** – The system is engineered to a sufficiently high standard to be safely placed in the hands of the users. The major product here is the Tested System. The DSDM process diagram does not show testing as a distinct activity because testing is happening throughout both the Functional Model Iteration

and the Design and Build Iteration. Some environments or contractual arrangements will require separate testing phases to be included at the end of the development of the increment, but this should not be the major activity encountered in more traditional approaches to development. Testing is just as important in DSDM and consumes just as much effort, but it is spread throughout development.

- **Implementation** – The cutover from the development environment to the operational environment. This includes training the users who have not been part of the project team. Iteration of the Implementation phase is applicable when the system is being delivered to a dispersed user population over a period of time.
  One product of this phase is the Increment Review Document. The Increment Review Document is used to summarize what the project has achieved in terms of its short-term objectives. In particular, it reviews all the requirements that have been identified during development and assesses the position of the system in relation to those requirements.

  The four possible outcomes are:

  - All requirements have been satisfied. Hence, no further work is currently needed.

  - A major area of functionality was discovered during development that had to be ignored for the time being in order to deliver on the required date. This means returning to the Business Study and taking the process on from there.

  - Lower priority functionality, which was known, had to be left out because of the timescale. This is now to be added, so the process returns to the Functional Model Iteration.

  - An area of lesser technical concern was omitted again due to time pressure, but can now be addressed by returning to the Design and Build Iteration.

- **Post-Project** – keeps the solution operating effectively. The iterative and incremental nature of DSDM means that maintenance can be viewed as continuing development. Maintenance is an expected part of a system's lifecycle, which can be accommodated using much the same approach as the initial development. Non-urgent fixes and enhancements may be batched up and implemented using DSDM techniques. This work should follow the DSDM principles with a high level of user involvement and should be treated as further iterations of the system, going round the DSDM method again starting with a quick pass through the Business Study.

DSDM has nine primary principles (http://www.dsdm.org):

- Active user involvement is imperative.

- The team must be empowered to make decisions.

- The focus is on frequent delivery of products.
- Fitness for business purpose is the essential criterion for acceptance of deliverables.
- Iterative and incremental development is necessary to converge on an accurate business solution.
- All changes during development are reversible.
- Requirements are base lined at a high level.
- Testing is integrated throughout the life-cycle.
- Collaboration and cooperation between all stakeholders is essential.

The benefits of using DSDM are:

- The users are more likely to claim ownership of the solution.
- The risk of building the wrong solution is greatly reduced.
- The final solution is more likely to meet the users' real business requirements.
- The users will be trained prior to deployment.
- The implementation of the business solution is more likely to go smoothly.

There are many advantages to DSDM. As in all the agile methods, there is increased user involvement, which can result in increased user satisfaction. DSDM uses many of the features of RAD which gives it a good base to build upon and can result in a time and cost savings as in RAD (Avison and Fitzgerald 2002).

The primary disadvantage to DSDM is that the DSDM consortium does not give free access to this framework, so there is a cost with learning and implementing this methodology. This also can result in increased complexity to develop the software using this method (Avison and Fitzgerald 2002).

2.14 Adaptive Software Development

"The adaptive model is built on a different world view. While cyclical like the evolutionary model, the phase names reflect the unpredictable realm of increasing complex systems (see Figure 2.14). Adaptive development goes further than its evolutionary heritage in two key ways. First, it explicitly replaces determinism with emergence. Second, it goes beyond a change in life cycle to a deeper change in management style. The difference can be subtle. For example, as the environment changes, those using a deterministic model would look for a new set of cause and effect rules, while those using the adaptive model would know that there are no such rules to find"(Highsmith 2000, 1997).

The adaptive software development model is based on the principles of complex adaptive systems (Highsmith 2000). "Complex Systems is a new field of science studying how parts of a system give rise to the collective behaviors of the system, and how the system interacts with its environment. Social systems formed (in part) out of people, the brain formed out of neurons, molecules formed out of atoms, the weather formed out of air flows are all examples of complex systems. The field of complex systems cuts across all traditional disciplines of science, as well as engineering, management, and medicine. It focuses on certain questions about parts, wholes and relationships. These questions are relevant to all traditional fields" (www.necsi.org). "Complexity theory helps us understand unpredictability and that our inability to predict does not imply an inability to make progress" (Highsmith 2002).

The theories of complex adaptive systems have recently been applied to business entities and more specifically business systems (Sutherland and Heuval 2002). Due to

the constraints naturally present in a business system, these systems make ideal candidates for the application of complex adaptive systems techniques (Sutherland and Heuval 2002).

Adaptive Software Development (ASD) is a software development method designed to apply the principles of complex adaptive systems concepts to the development of business systems. "The practices of ASD are driven by a belief in continuous adaptation – a different philosophy and a different life cycle geared to accepting continuous change as the norm. In ASD, the static plan-design-build life cycle is replaced by a dynamic speculate-collaborate-learn life cycle. It is a life cycle dedicated to continuous learning and oriented to change, re-evaluation, peering into an uncertain future, and intense collaboration among developers, management, and customers" (Highsmith 2002).

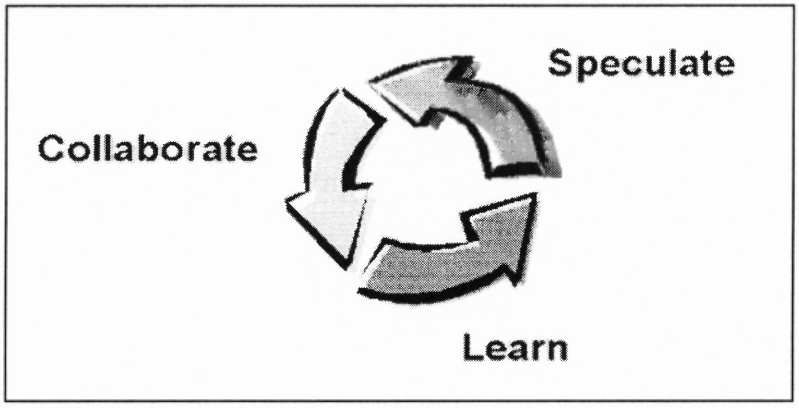


Figure 2.14 Phases of the adaptive model.
Source: Highsmith 2000

Figure 2.14 shows the phases of the adaptive model. "Instead of the widely accepted loop to express iterative development – *plan, build, revise* – Highsmith proposes *speculate, collaborate, learn*" (Kruchten 2001). The phases can be explained as follows:

- Speculate – "According to CAS theory, outcomes are unpredictable. Yet wandering around, endlessly experimenting on what a product should look like is not likely to lead to profitability either. 'Planning', whether it is applied to overall product specifications or detail project management tasks, is too deterministic a word. It carries too much historical baggage. 'Speculate' is offered as a replacement" (Highsmith-http://www.adaptivesd.com). "Speculating is analogous to fuzzy planning" (Highsmith 2000).

- Collaborate – "means working together to produce a shared result. It is an act of adding value to a product" (Highsmith 2000).

- Learn – "In an adaptive environment, learning challenges all stakeholders, including both developers and customers, to examine their assumptions and use the results of each development cycle to learn the direction of the next" (Highsmith-http://www.adaptivesd.com).

The six basic characteristics of an adaptive lifecycle are mission focused, component based, iterative, time boxed, risk driven, and change tolerant (Highsmith 2000).

- Mission focused – Although the final results may be fuzzy in the initial phase, the overall mission is well defined.

- Component based – Groups of features are developed (i.e., results, not tasks, are the focus).

- Iterative – Emphasis is placed on "re-doing" as much as "doing."

- Time boxing (i.e., setting fixed delivery times for projects) – Time boxing forces ASD project teams and their customers to continuously re-evaluate the project's mission, scope, schedule, resources, and defects.

- Risk driven - Similar to the spiral development model, adaptive cycles are guided by the analysis of critical risks.

- Change tolerant – The ability to incorporate change is viewed as a competitive advantage (not as a problem).

The benefits of ASD include the following (Highsmith 2000):

- Applications are a closer match to customer requirements due to constant evolution.
- Changing business needs are easily accommodated.
- The development process adapts to specified quality parameters.
- Customers realize benefits earlier.
- Risk is reduced.
- Confidence is established in the project early on.
- It forces developers to more realistically estimate their ability. Lack of complete knowledge is assumed and comprehensive feedback mechanisms are built to compensate.

2.15 Open Source Software Development

"While not classifying the Open Source Software (OSS) development approach as a silver bullet (Brooks 1987), the case can be made that OSS addresses many aspects of the software crisis, in that reliable, high quality software may be produced quickly and inexpensively" (Feller 2000).

"The basic idea behind open source is very simple: When programmers can read, redistribute, and modify the source code for a piece of software, the software evolves. People improve it, people adapt it, and people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing" (www.opensource.org 2003).

Open Source Software is software released under a license conforming to the Open Source Definition (OSD), as articulated by the Open Source Initiative (Feller 2002). The Open Source Definition is a bill of rights for the computer user (Perens 1999). Table 2.3 lists the tenets of the latest revision (1.9) of the OSD (www.opensource.org 2003).

Table 2.3 Open Source Definition (http://www.opensource.org)

Introduction

Open source doesn't just mean access to the source code. The distribution terms of open-source software must comply with the following criteria:

1. Free Redistribution

The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.

2. Source Code

The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.

3. Derived Works

The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

4. Integrity of The Author's Source Code

The license may restrict source-code from being distributed in modified form *only* if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

5. No Discrimination Against Persons or Groups

The license must not discriminate against any person or group of persons.

6. No Discrimination Against Fields of Endeavor

The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

7. Distribution of License

The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

8. License Must Not Be Specific to a Product

The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.

9. The License Must Not Restrict Other Software

The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.

***10. The License must be technology-neutral**

No provision of the license may be predicated on any individual technology or style of interface.

Open Source Software is developed using Open Source Software Development. This methodology has been characterized as "extreme distributed software development" (Mockus 2002) or "massive parallel development" (Feller 2000). In this paradigm, developers are geographically separated, rarely or never meet face to face, and coordinate their activity by means of an electronic medium (i.e., Internet, email or bulletin boards) (Mockus 2002).

Open source development is a development process that is radically different, according to OSS proponents, from the usual industrial style of development. The main differences most often mentioned are the following (Mockus 2002):

- OSS systems are built by potentially large numbers (i.e., hundreds or even thousands) of volunteers.
- Work is not assigned; people undertake the work they choose to undertake.
- There is no explicit system-level design, or even detailed design.
- There is no project plan, schedule, or list of deliverables.

The typical OSS project has a single lead developer (or a small core of developers as in the Apache case) who takes the lead in establishing the project direction. Co-developers, having full access to the source code, submit code patches to solve various problems that arise and to add to the functionality of the software (Feller 2000). This type of development as been analogized to a Bazaar where there is much group interaction versus a Cathedral where laborers work silently alone (Raymond 1998).

There are generally no documented norms or procedures for OSS development projects. However, there are many customs and taboos that are associated with the

process. One such taboo is a practice called "forking" whereby the project develops into incompatible strands (Feller 2000).

There are many of tools and practices that are used in Open Source Development. These tools are usually specific to the project's requirements and culture. A version control system, such as CVS, is used to maintain code, through which anyone can browse, however, usually only core developers can commit code. Bugs and feature requests are tracked by means of an issue tracking system such as Bugzilla.

Besides these code management tools, open source software projects use a number of tools for collaboration and coordination. The primary ones are group mailing lists, asynchronous discussion forums and more recently, chat facilities (Ankolekar, Herbsleb, Sycara 2003).

In many ways, Open Source Development can be categorized as an agile method (Warstaa and Abrahamsson 2003). It shares many characteristics of agile development in that it is incremental, cooperative, straightforward, and adaptive (Warstaa and Abrahamsson 2003). A primary distinguishing characteristic of Open Source from the Agile methods is that electronic collaboration is used in place of the direct interaction required by many of the agile methods (Augustin et al. 2002).

There are tools being developed that facilitate electronic collaboration in a programming environment. One such tool is Coven (Chu-Carroll and Sprenkle 2000). Coven is a tool that allows programming teams to coordinate their work by providing a mechanism that allows programmers to communicate and organize their work in a fashion that matches the development environment (Chu-Carroll et al. 2000).

Open Source Software has been very successful based on market share, reliability, performance, scalability, security, and total cost of ownership (Wheeler 2003). Popular OSS includes the second most popular operating system (Linux), the most popular web server (Apache), the web domain name serving system (Bind), and the leading e-mail server (Sendmail) (Wheeler 2003). However, Open Source Software seems to be especially suited to large projects such as operating systems, utilities, and network tools (Feller 2000).

The challenge facing Open Source Software development appears to be how to make it work in the enterprise (Augustin et al. 2002) and in commercial development (Warstaa and Abrahamsson 2003).

2.16 Classifying Development Methods

Prior research has attempted to classify the various software development methodologies according to varying schema (Avison and Fitzgerald 2003, Scacchi 2001). There are several diverse classification models in the current body of research. This paper will introduce some of them and offer a model of its own.

One way of classifying development methodologies is by era. It has been stated that software development methodologies have progressed through several distinct eras (Avison and Fitzgerald 2003). There was the "pre" methodology era when no methods were used, the "early" methodology era marked by the creation of the life cycle methodologies, and the methodology era when several new and distinct methods emerged. The methodology era gave rise to methods such as Spiral, RAD, JAD, prototyping, and Object-Oriented.

The argument can be made that the current state of software development can be characterized as "post" methodology (Avison and Fitzgerald 2003). In the "post" methodology era, "Real-world performance has led some developers to reject methodologies in general terms and attack the concepts (such as step-by-step development and meticulous documentation) on which they are based" (Avison and Fitzgerald 2003). This idea is supported by the "agile manifesto" (Cockburn 2002) and the current array of agile methodologies.

Another schema for classifying development methodologies is by focus of attention. A classification model is offered by Scacchi (2001), in which there are at least three alternative sets of models of software development, "Lifecycle" based, "Product Development" based, and "Process" based. Lifecycle based methods would be the

traditional "waterfall" method, as well as an array of mutations such as stepwise refinement, incremental development and release, industrial and military standards, and the capability models. Scacchi (2001)

Product development models seek to enable the creation of executable software implementations either earlier in the development process or more rapidly. Scacchi (2001). Examples of this type of method are prototyping, JAD, RAD, reusable components, application generation, and evolutionary models. Scacchi (2001)

The final category of the Scacchi classification system is Process Models. Process models "often represent a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution" Scacchi (2001). Scacchi further dissects process models into operational and non-operational. Non-operational process models denote conceptual processes while operational process models can be viewed as scripts or programs (Scacchi 2001). An example of a non-operational process model would be Boehm's spiral method (Boehm 1988). Operational models encompass rapid prototyping, software automation, and software process automation and programming (Scacchi 2001).

Table 2.4 offers a classification schema for the methods covered in this paper according to the main emphasis of their methodological process. The table identifies four main areas of emphasis. Those areas are:

- Lifecycle – the degree to which the methodology adheres to the traditional lifecycle paradigm.
- Iteration – the degree to which the methodology follows an iterative paradigm.
- Adaptive – the degree to which the methodology emphasizes adaptation (i.e., evolution) of the system.

- Collaborative – the degree to which the methodology emphasizes the collaborative effort of the group or the "people" factors in software development.

The traditional waterfall approach is all about following the lifecycle approach (Royce 1970) and is thus reflected as such in Table 2.4. It has the highest number of degrees in the lifecycle column of any of the methods, and the lowest number of degrees in any of the other columns.

The prototype method, while essentially a lifecycle approach, begins to expand its focus into some of the other areas. For instance, evolutionary prototyping has a degree of iteration and a degree of adaptation to it. Often times, the initial prototype is developed and then iteratively adapted until it closely resembles the required system.

The spiral method shifts the focus from following the traditional lifecycle approach to following an iterative, incremental approach (Boehm 1988). It still allows for potentially adhering to a lifecycle approach, particularly within an iterative cycle. Any given iteration with the spiral method may appear to be a traditional waterfall type methodology. The iterative nature of the spiral method gives it a degree of the adaptive property. It is not adaptive in the sense of one of the agile methods, due mostly to its high emphasis on planning and adhering to a prescriptive method throughout the iterative cycle. However, it is adaptive in that each cycle assesses current conditions (as well as risk factors) and adjusts accordingly (Boehm 1988).

Joint application development is also primarily a lifecycle approach. However, it also places a large emphasis on collaboration (Wood et al. 1995) The collaborative focus in joint application development is usually centered in the design phase, even though it can be re-visited during any phase of the life cycle. Even though the collaboration in

JAD is not as omnipotent as in a purely collaborative method such as open source development, it still is strong enough to earn JAD a degree of emphasis in Table 2.4.

Rapid Application Development also earns the full measure of lifecycle emphasis in Table 2.4. The primary focus of RAD is to take the traditional lifecycle and shorten it in order to effectively deliver a software product in less amount of time (Martin 1991). However, RAD is also similar to joint application development in that is places a strong emphasis on the collaboration and the "people" factors of a project (Hoffer 1999). Furthermore, rapid application development also has a degree of iterative methodology to it (Agarwal et al. 2000). Often times in RAD, shortened lifecycles are performed iteratively until the project is completed.

Object Oriented Development does not map as easily into the categories of Table 2.4 as some of the other methodologies. This is because OOD can take on many forms and emphases. It is definitely a highly structured "software-engineering" process (Kruchten 2000) and thus must earn some degree of lifecycle points in Table 2.4. However, the primary emphasis of OOD must be characterized as iterative in nature. The chief pillar of OOD is in its definition of a project in general terms and then delving deeper through successive refinements (Leffingwell 2001) much like peeling an onion (Hoffer 1999). This iteration process also gives object-oriented development a degree of adaptability. Each iteration gives the developers time to assess the current conditions and adapt the project accordingly.

All of the agile methodologies will have certain agile characteristics as defined by the agile manifesto (Beck et al. 2001). They all, by nature, are iterative, adaptive, and collaborative. The nuances distinguishing the agile methods lie mostly not in theory, but

in their practical application. However, some of the agile methods also put more weight into planning and other prescriptive, lifecycle like activities.

Extreme Programming is a typical agile method in that it emphasizes adaptation over planning (Beck 2000). Even though it includes some elements of the lifecycle such as planning and testing, it is so far removed from the lifecycle that it does not warrant any degree of emphasis in Table 2.4. Also, even though it receives a high degree of emphasis points for collaboration, it does not receive the full measure as it lacks the same degree of collaboration as open source development.

Crystal receives similar rankings for iteration, adaptation, and collaboration as the other agile methods. However, crystal allows and plans for lifecycle activities. It is "non-jealous" in that it can incorporate other methods (Cockburn 2002). Crystal also comes in "flavors" from light to heavy depending on the appropriate circumstance (Cockburn 2002). The heavier versions are similar to traditional lifecycle approaches, and thus why crystal earns a degree of lifecycle emphasis.

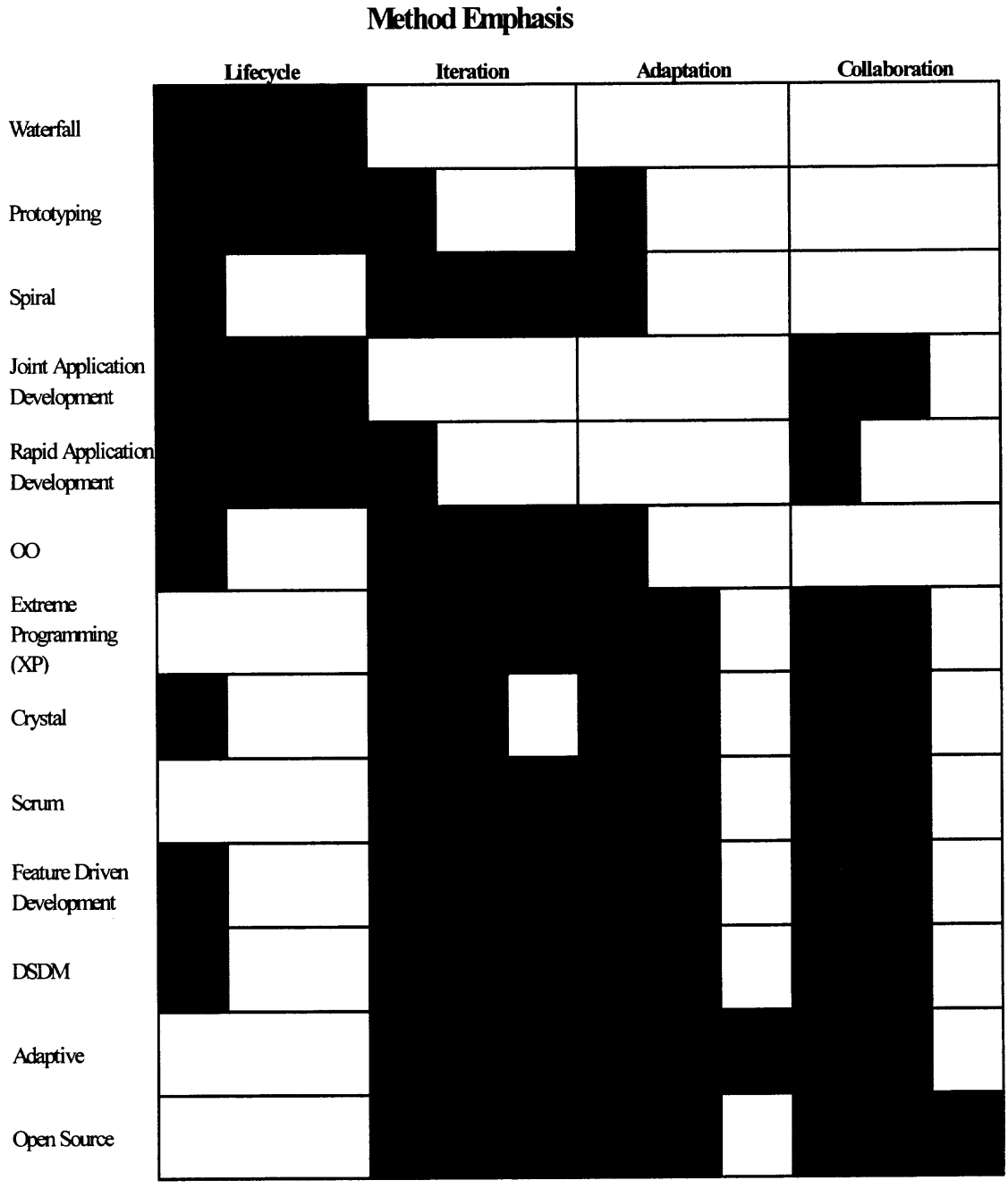
Scrum is similar to extreme programming in that it bears little resemblance to the traditional lifecycle. However, it does encompass all of the traditional emphases of the agile methods. Feature driven development and DSDM, on the other hand, are the closest to the lifecycle methodologies. In fact, FDD is promoted as combining the best of both the heavy and agile worlds (Coad 1999, Palmer 2002) while DSDM has a series of stages similar to those in the traditional lifecycle (http://www.dsdm.org). FDD and DSDM, therefore, score degrees of emphasis in the lifecycle category.

Adaptive software development is probably the "lightest" of all the agile processes. Therefore it has no degrees in the lifecycle column of Table 2.4. It, like

scrum, is purely an iterative, adaptive collaborative process. However, the process of adaptive software development makes it the most adaptive of all the methodologies.

Open Source Development, while not officially characterized as an agile method is in many ways similar (Warstaa and Abrahamsson 2003). It is iterative and adaptive like an agile method and is thus scored high points in those areas. However, open source by its very definition, is about collaboration and as such is scored the highest degree in the collaboration column in Table 2.4.

Table 2.4 A Comparison of Covered Development Methods by Emphasis



2.17 Contingent Factors and Method Engineering

There are clearly several alternative methods available. The question that still remains unanswered is, "Which one is best?" Research shows that the answer to that question may not be a simple one. In fact, research has shown that no one method can claim to be the best method for every software project (Cockburn 2002, Fitzgerald et al. 2003).

One solution that has been offered recently is the "Contingency" or "Contingent Factors" Approach (Avison and Fitzgerald 2003, Fitzgerald et al. 2003). With this approach, the methodology used is contingent upon several factors inherent to the project. These factors would include: "the type of project and its objectives, the organization and its environment, the users, and the developers and their respective skills. The type of project might also differ as to purpose, complexity, structure, degree of importance, projected life, and potential contribution to overall corporate performance. Different environments might exhibit different rates of change, number of users affected by the system, user skills, and analyst skills" (Avison and Fitzgerald 2003).

One area of concern for the contingency approach has been the fact that it may not be feasible or possible for all the developers in an organization to be familiar with all of the possible methods that would work best for a given situation (Fitzgerald et al. 2003). A suggested alternative has been a technique called "Method Engineering" (Brinkkemper 1996, Fitzgerald et al. 2003). With this technique, a method is constructed from "existing discrete predefined and pre-tested method fragments" (Fitzgerald et al. 2003). Using a method-engineering tool, software developers build a meta-method that is made up of fragments from popular development methodologies. The fragments are each designed to handle a particular contingency inherent to the software project.

Figure 2.15 shows a typical method-engineering model. A repository holds all the method fragments. The developers pick the fragments from the repository to build the meta-model.

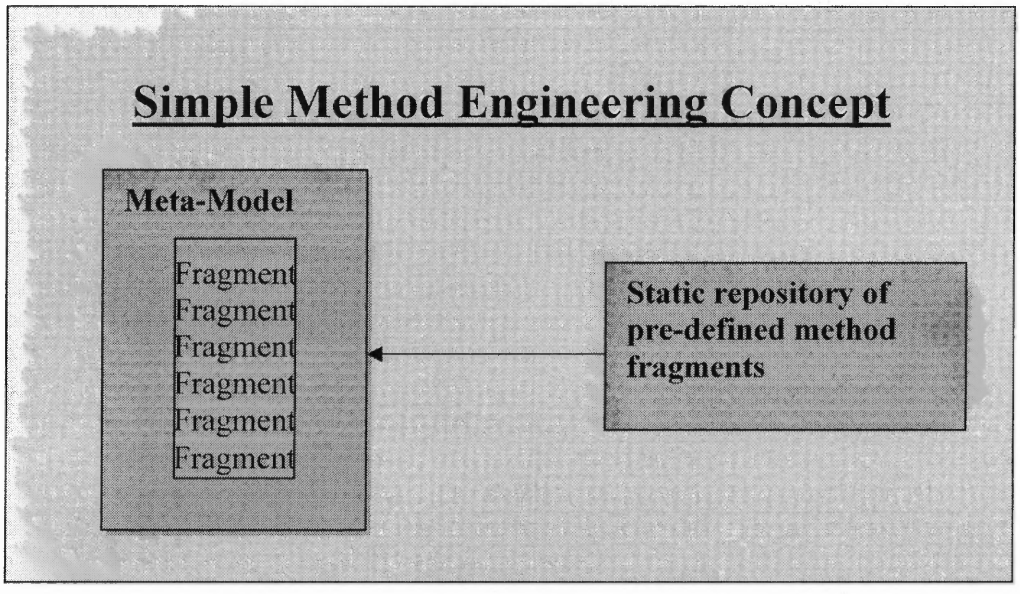


Figure 2.15 A typical method engineering process.

There are several shortcomings to the typical method-engineering model:

1. It is impossible to plan for every contingency that may come up during the development of the software product and therefore critical fragments will always be missing (Rossi 2000).

2. The repository of pre-defined method fragments is somewhat static, and can only be changed by the developers of the ME tool.

3. The combination of problems 1 and 2 give the typical ME tool a prescriptive nature, rather than an adaptive one. The tool does not adapt well to unforeseen circumstances.

The contingency factors approach and method engineering have two things in common:

1. Both approaches match the best methodology to the project.
2. Both approaches provide an explicit process to choose the best methodology (or method fragment).

Two commonalties can be considered as indicators of a method tailoring approach when taken in their totality. Some methodologies such as RUP and crystal offer a mechanism to tailor the methodology to the project. This leads to the question as to whether this type of approach should be considered a method tailoring approach. In the case of RUP, the answer is clear. RUP is a methodology and not a method tailoring approach. The difference is this. Bona fide method tailoring approaches such as contingency factors and method engineering allow for the incorporation of techniques, processes, and tools, etc. that exist outside the realm of a specific methodology. While RUP allows itself to be adjusted to the properties of the project, it does not explicitly allow for other methodologies (or fragments thereof) to be incorporated into the process and therefore does not always allow for the best methodology to be matched to the project.

The crystal methodology offers a different perspective. It touts itself as a "non-jealous" (Cockburn 2002) methodology that encourages the incorporation of other methodology fragments. This means that it surpasses the test that excludes RUP as a method tailoring approach. Yet it stills fails the second test of a method tailoring approach. A true method tailoring approach will provide a means by which to use the best methodology or fragment to meet the given the circumstances of the project. While Crystal offers a two dimensional scale upon which to measure the project and thus adapt itself, it does not offer an explicit vehicle which can be used to pick the best methodology

fragment from outside its realm. Therefore crystal does not pass test number two and cannot be considered a method tailoring approach.

2.18 Conclusions

This state of the art review of the most popular software development methods leads the researcher to several conclusions. Clearly, each of the methods described in this paper has its advantages and disadvantages (See Table 2.5 for a summary). Each method has a group of dedicated followers who firmly believe in the superiority of their method. However, the fact remains, that there is still no "silver bullet" to the software crisis.

There is further evidence that those that continue to use the traditional development methods in practice, find ways to adapt those processes to their situation. There is often a wide disparity between the official rules of the method and the actual implementation of the method by developers in the real world (Fitzgerald et al. 2003).

The research has pointed out three distinct areas of concern:

1. No ONE method can claim to be the best method for EVERY software project (Cockburn 2002, Fitzgerald et al. 2003).
2. Present day developers are becoming increasingly discouraged with the traditional methods and their shortcomings (Avison and Fitzgerald 2003).
3. Those developers that have continued to use the traditional methods have modified and adapted those methods to meet the specificities of the project (Fitzgerald 1997, Fitzgerald et al. 2003).

The challenge, therefore, is to find a way to tailor the methodology to the project in such a way that the best method is always used for the correct project, and at the correct phase in the project, and that the method itself is not so cumbersome that it becomes more tedious than the project itself. As Alistair Cockburn puts it: "The mystery

is how to construct a different methodology for each situation, without spending so much time designing the methodology that the team doesn't deliver software. You also don't want everyone on your project to have to become a methodology expert" (Cockburn 2002). The Contingent Factors approach and Method-Engineering appear to be making progress in this area, but still have their shortcomings (Fitzgerald et al. 2003, Rossi 2000).

One important factor that cannot be ignored is the practical application of these ideas in the real world. Ideas such as "Contingent Factors" and "Method Engineering" are theoretical in nature and have not been studied extensively in real development practice (Fitzgerald et al. 2003). Future research must find a way to bridge this disparity between academic research and the world of the practitioner.

Future research must also focus on the people factor. It is becoming more and more apparent that the effectiveness of methods lies in the people and not in the process (Baskerville, Levine, Pries-Heje, Ramesh, and Slaughter 2002, Cockburn 1999). People are "highly variable and non-linear" (Cockburn 1999) and need to be recognized as a primary variable in the success or failure of a project. Also, it appears from the research conducted in this paper, that the success being enjoyed in the open source community cannot be ignored. Collaboration appears to be a powerful tool that must be incorporated into any future methods or meta-method.

Finally, future research needs to be done on how to blend the many factions of software development methods that exist (as exhibited in this paper). Method Engineering appears to be a strong step in this direction, but has some apparent flaws. Future techniques must find a way to draw upon the strengths of the many methods.

How to draw upon the prescriptive features of the "heavy" methods, yet fall back on the adaptive features of the agile methods? In other words, in any given project, prescribe what can be prescribed, and adapt to or problem solve the rest.

Table 2.5 A Summary of Covered Development Methods

Method	Author	Type	Characteristics	Advantages	Disadvantages
Waterfall	?	Heavy	Project is divided into distinct phases	Ability to quantify an abstract subject. Especially useful to management.	Costly if req. change. No product until late in the process
Prototype	?	Heavy	Develop an initial model of the system for user feedback	Allows the user to provide feedback before a large investment made	Lack of documentation and poor project visibility
Spiral	Boehm	Heavy	Successive refinement of requirements and design	Incorporates risk	Can incur endless fiddling
Joint Application Development	IBM	Heavy	Workshops where users and I.S. people meet to define system	Brings important participants of a project together to hash out the details	High Cost and time consuming
Rapid Application Development	Martin	Heavy	Development life cycle designed for faster development and higher quality	Quicker development and less development costs	Higher risk of low quality systems.
OO	Coad, Yourdon, Robinson, Booch	Heavy	Decompose a problem into objects that encapsulate data and behavior	Better modeling, analysis & design. Improved communication	Increased development time. Poorer runtime performance.
Extreme Programming (XP)	Jeffries Beck	Agile	Short cycles, incremental planning, flexible schedule	Increased customer involvement, teamwork, & communication	Code centered vs. design centered. Lack of documentation
Crystal	Cockburn	Agile	People-centric, ultra-light, shrink to fit	Takes into account current cultural conditions	Requires constant fine tuning.
Scrum	Schwaber	Agile	Lightweight, iterative, incremental process	Increased productivity and adaptability	Poor resource estimating. Limited to small teams.
Feature Driven Development	Deluca Coad	Agile	Design and Build by feature Model-driven short-iteration process.	More planning then rest of agile methods	Increased complexity
DSDM	DSDM Consort	Agile	Works under time constraints of RAD	Increased user satisfaction. RAD time and cost savings	Requires special training to use (not free). Complex.
Adaptive	Highsmith	Agile	Based on complex adaptive systems. Emergence vs. determinism	Adaptive to change.	Poor resource estimating.
Open Source	Torvalds	?????	Collaborative Development	Increased quality and reliability.	Adapting the process to business/commercial applications

CHAPTER 3

A COLLABORATIVE HIERARCHICAL INCREMENTAL PROBLEM SOLVING MODEL

3.1 Theoretical Foundation

The goal of this research is to offer a solution to the problems described in Chapter 2 in the form of a model. It is the aim of this model to be inclusive of the various IS development methodologies and to serve as a guide to IS development. In order to accomplish this, a paradigm must be employed in order to properly create this model. This paradigm can be found in the arena of design science theory.

The design-science paradigm seeks to create things that serve human purposes (March and Smith 1995). It extends the boundaries of human and organizational capabilities by creating new and innovative artifacts (Hevner, March, Park, and Ram 2004). IT artifacts are defined as *constructs* (vocabulary and symbols), *models* (abstractions and representations), *methods* (algorithms and practices), and *instantiations* (implemented and prototype systems) (Hevner et al. 2004). Design science is used to create and evaluate IT artifacts that are intended to solve identified organizational problems (Walls, Widmeyer, and El Sawy 2004, Hevner et al. 2004).

The IT artifact has been the focus of much debate within IS research circles in recent years (Benbaset and Zmud 2003, Argawal and Lucas 2005, Robey 2003, Ives, Parks, Porra, and Silva 2004, DeSanctis 2003). The majority of research in the IS field has not focused on IT artifacts themselves, but rather on the context, the capabilities, or effect of those artifacts on the organizations, people, and processes upon which they are applied (Orlikowski and Barley 2001). In their 2003 article, Benbaset and Zmud suggested that the IT artifact and its immediate "nomological net" (i.e., capabilities,

practice, usage, and impact) are under investigated in IS research (Benbaset et al. 2003). They also suggested that IS research suffers from errors of inclusion (i.e., focusing on constructs that lie outside the IT artifact and it nomological net) and errors of exclusion (i.e., disregarding constructs that reflect the core properties of the IS discipline) (Benbaset and Zmud 2003).

The vast majority of IS research that has been conducted draws its theoretical foundation from the theories of the behavioral sciences which are based on philosophies from the natural and social sciences (Hevner et al. 2004, Walls et al. 2004). Behavioral science research is concerned with the development and justification of theories that explain or predict behavior by seeking the "truth" while design science seeks to create utility by creating "what is effective" (Hevner et al. 2004). With regard to technology and artifacts, the behavior science research paradigm is *reactive* in the sense that it takes the technology as a given. Design science research can be considered *proactive* in its relationship to technology as it seeks to create and evaluate innovative IT artifacts (Hevner et al. 2004). However, it can be argued that design theories are based on the natural and social science theories, as design theories must operate within the laws of those theories (Walls et al. 2004).

The design science concept was first prominently introduced to the IS field by a 1992 article titled "Building an Information Design Theory for Vigilant EIS" that appeared in *Information Systems Research.* A follow-up article by the original authors twelve years later showed that an exiguous number (26) of articles had cited the original article (Walls et al. 2004). The reasons for the tepid response are not evident and can range from usability issues (Walls et al. 2004) to lack of interest on the part of

researchers (Orlikowski et al. 2001). However, there has been a growing interest in and a call for an increased focus on design science research in the IS field (Orlikowski et al. 2001, Hevner et al. 2004, Walls et al. 2004). While design science research has not been embraced in the IS field (Walls et al. 2004, Orlikowski et al. 2001), it has been utilized extensively in other disciplines such as engineering and the sciences of the artificial. (Hevner et al. 2004).

Several authors have presented frameworks and guidelines for design science research (March and Smith 1995, Hevner et al. 2004, Walls et al. 2004). The earliest significant framework was introduced by Walls et al. and is presented in Table 3.1. Walls et al. classify design theory components into either design products or design processes. Components of each class are then defined in an ordinal fashion after a kernel theory is drawn from the natural or social sciences.

Table 3.1 Components of an Information System Design Theory (Walls et al. 2004)

Design Product		
1.	Meta-requirements	Describes the class of goals to which the theory applies
2.	Meta-design	Describes a class of artifacts hypothesized to meet the meta-requirements
3.	Kernel theories	Theories from natural or social sciences governing design requirements
4.	Testable design product hypotheses	Used to test whether the meta-design hypotheses satisfies the meta-requirements
Design Process		
1.	Design method	A description of procedure(s) for artifact construction
2.	Kernel theories	Theories from natural or social sciences governing design process itself
3.	Testable design process hypotheses	Used to verify whether the design hypotheses method results in an artifact which is consistent with the meta-design

Hevner et al. (2004) list seven guidelines for performing design science research. The guidelines are not mandatory but should be addressed for a design science research project. Those guidelines are:

1. Creation of an innovative, purposeful artifact
2. A specified problem domain
3. Thorough evaluation of the artifact
4. The artifact must solve an unsolved problem in an innovative way
5. The artifact must be rigorously defined, formally presented, coherent, and internally consistent.
6. A problem space must be constructed and a mechanism is enacted to find a solution.
7. The results of the research must be communicated effectively.

The remainder of this chapter will proceed in that format and will address each topic accordingly.

3.2 The Model

Guideline # 1: A Specified Problem Domain Must Be Defined

Researchers and practitioners commonly seek ways to improve system analysis and development through methodological advances. Unlike traditional science wherein a methodology is the study of a constituent group of methods used to solve a problem, a methodology in software engineering and IS development has a different meaning. A common definition suggests a methodology is a recommended collection of phases, procedures, rules, techniques, tools, documentation, management, and training used to develop a system (Avison and Fitzgerald 2003, Cockburn 2002, Hoffer et al. 2005). There have been significant advances and changes to methodological techniques over the last 30 years and those changes can be characterized into specific eras (Avison and Fitzgerald 2003, Fowler 2002).

The advent of codified methodologies (e.g., Waterfall) marked the early methodology period. During this era many new methodologies were introduced. Methodologies from this era can be classified according to significant approaches such as structured (Yourdon and Constantine 1979), data-oriented, process-oriented, prototyping (Naumann 1982), participative (Mumford 1981), and object-oriented (Coad and Yourdon 1991).

People argue that in recent years we have entered a post-methodology era wherein researchers and practitioners are questioning the older methodological philosophies (Avison and Fitzgerald 2003, Fowler 2002). Most of the serious criticism of the prior approaches is that they are bureaucratic and labor intensive or "heavy" methodologies (Fowler 2002). In response to this, new methodologies introduced in "post-

methodology" period are considered as lightweight or agile methodologies because of the lack of bureaucratic contingencies inherent to the heavy methodologies (Fowler 2002). These agile methodologies are considered by many in this postmodern era to be "amethodological" (i.e., a negative construct connoting not methodological) (Truex et al. 2000). The biggest criticism of the agile methodologies has been the lack of empirical evidence supporting the claims of their benefits and their lack of theoretical foundation (Abrahamsson et al. 2003). However, there is a growing body of literature both supporting and repudiating the claims of success of the agile methodologies (Abrahamsson et al. 2003, Choi and Deek 2002).

The research has pointed out three distinct areas of concern:

1. No one methodology can claim to be the best method for every software project. (Cockburn 2002, Fitzgerald et al. 2003)
2. Present day developers are becoming increasingly discouraged with the traditional methods and their shortcomings (Avison and Fitzgerald 2003).
3. Those developers that have continued to use the traditional methods have modified and adapted those methods to meet the specificities of the project (Fitzgerald 1997, Fitzgerald et al. 2003).

One concept that has shown much promise is the idea of tailoring a methodology to the actual project development context (Fitzgerald et al. 2003). The contingency factors approach suggests that specific features of the development context should be used to select an appropriate methodology from a portfolio of methodologies. This approach requires developers to be familiar with every contingent methodology or have contingency built in as part of the methodology itself.

A suggested alternative has been a technique called "Method Engineering" (Brinkkemper 1996, Fitzgerald et al. 2003). With this technique, a methodology is

constructed from a repository of "existing discrete predefined and pre-tested method fragments." (Fitzgerald et al. 2003). Using a method-engineering tool, software developers build a meta-method that is made up of fragments from popular development methodologies. The fragments are each designed to handle a particular contingency inherent to the software project. The fragments are categorized as either *product* or *process*. Product fragments are artifacts capturing the structure in deliverables such as diagrams, tables, or models, while process fragments project strategies and detailed procedures (Brinkkemper 1996).

ME has several shortcomings. For example, it is impossible to plan for every contingency that may arise, and therefore, critical fragments will always be missing (Rossi, Ramesh, Lyytinen, and Tolvanen 2004). Also, the burden of selecting the correct fragment falls upon the analyst, possibly aided by a ME tool (Truex et al. 2000), but ME tool development is a problematic procedure (Fitzgerald et al. 2003). Thus, evolution of software development methodologies using fragments is problematic.

Both contingency factors and ME techniques have had little success in practical industry applications (Fitzgerald et al. 2003, Rossi et al. 2004). However, ad hoc methodology tailoring has been an implied concept for many years in industry (Fitzgerald 1997) and many popular methodologies such as the Rational Unified Process (Jacobson et al. 1999) nearly mandate tailoring through modification and adaptation in order for them to work in a specific context.

Guideline #2: A problem space must be constructed and a mechanism is enacted to find a solution

In accordance with this guideline, the Walls et al. (2004) framework can be applied. Figure 3.1 shows a flowchart from Walls et al. demonstrating the relationships among the components of an IS design theory. The Walls model distinguishes between products, ("a plan of something to be done or produced") and processes, ("to so plan and proportion the parts of a machine or structure that all requirements will be satisfied" Walls et al. 2004). The goal is the creation of a model. Model artifacts are used to abstract and represent phenomena (Hevner et al. 2004). A model would fall under the category of a product, and thus, the steps of creating testable design product hypotheses can logically be followed.

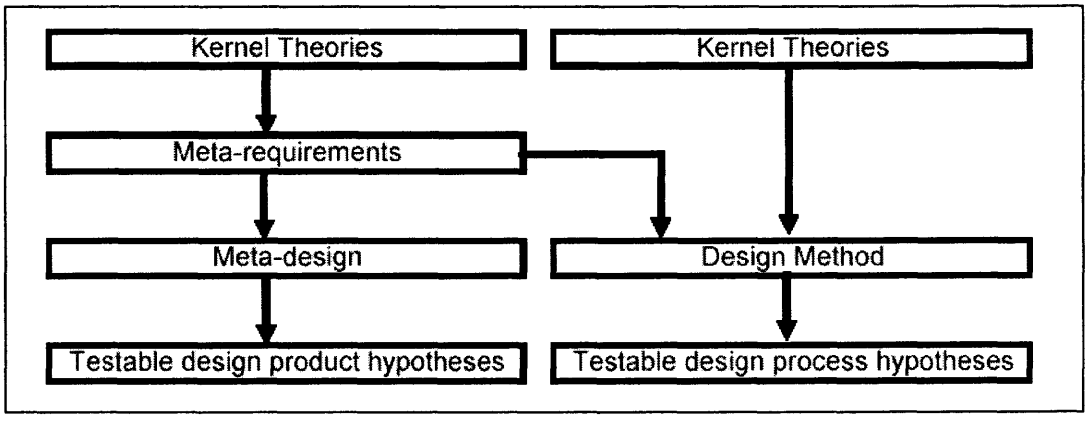


Figure 3.1 Relationships of IS design theory components.
Source: Walls et al. 2004

The process begins by choosing a kernel theory from the natural or social sciences. Walls et al. posited that design theories should be based on theories from natural or social science theories since the "laws" of the natural and social world govern the components that comprise an information system (Walls et al. 2004). The kernel theory for the model will be General Systems Theory General systems theory is an

interdisciplinary field that studies systems as a whole. It focuses on the complexity and the interdependence of the parts of a system. "General Systems Theory is a name which has come into use to describe a level of theoretical model-building which lies somewhere between the highly generalized constructions of pure mathematics and the specific theories of the specialized disciplines." (Boulding 1956)

Hungarian biologist Ludwig von Bertalanffy originally proposed general systems theory in 1928 (von Bertalanffy 1928) as a reaction against the reductionistic and mechanistic approaches to scientific study, and in an attempt to unify the fields of science. From the work of Descartes, and with significant influence from Bacon and Newton, the "scientific method" had progressed under the assumptions that an entity could be broken down into its smallest components so that each component could be analyzed independently (reductionism), and that the components could be added in a linear fashion to describe the totality of the system (mechanism).

Von Bertalanffy proposed that both assumptions were wrong. Rather than reducing an entity to the properties of its parts or elements, general systems theory focuses on the arrangement of and relations between the parts that connect them into a whole (holism). "It is necessary to study not only parts and processes in isolation, but also to solve the decisive problems found in organization and order unifying them, resulting from dynamic interaction of parts, and making the behavior of the parts different when studied in isolation or within the whole..." (von Bertalanffy 1969).

General systems theory can be thought of as being an inherently dichotomous theory (Umpleby 2001). One part of general systems theory represents von Bertalannfy's original intent in developing the theory. The idea was to define a theory

that could identify the isomorphic laws across diverse disciplines in an attempt to integrate the various scientific disciplines (von Bertalannfy 1969). From this viewpoint general systems theory can be thought of as an abstraction of several previous theories (Umpleby 2001) and thus a model building theory (Boulding 1956).

The second part of general systems theory began as its "proof", but later evolved as a basis for several subsequent theories, laws, and disciplines. For instance, cybernetics, complex adaptive systems, and the law of requisite variety are offshoots of this part. This part added a new dimension to scientific study by demonstrating the application of the theory.

For this research, both schools of general systems theory will be applied. The model building part is used to define the isomorphic principles across the various system development methodologies and form the basis of the model and the application of the theory is used to enforce the model's value.

The goal of general systems theory was to find common ground upon which scientific study could be conducted across all disciplines. Von Bertalanffy felt that this could be accomplished not by breaking entities down, but by finding laws that were common among the various fields. "A unitary conception of the world may be based, not upon the possibly futile and certainly farfetched hope finally to reduce all levels of reality to the level of physics, but rather on the isomorphy of the laws in different fields" (von Bertalanffy 1969).

Von Bertalanffy defined a system as "complexes of elements standing in interaction". He found that conventional physics dealt only with closed systems (i.e., systems which are isolated from their environment). In particular, the laws of

thermodynamics expressly stated that they were intended for closed systems. The essence of the second law of thermodynamics (law of entropy) is that entropy (i.e., the degree of disorder or uncertainty in a system (von Bertalanffy 1969) will increase over time in a closed system.

General systems theory realizes that many systems, by their nature, are open systems that interact with their environment. Von Bertalanffy observed that the second law of thermodynamics does not hold true in open systems. He realized that in an open system, the degree of disorder or uncertainty *decreases* over time or that "negative entropy" occurs (von Bertalanffy 1969).

General systems theory also realizes that open systems have a tendency to self-organize. This is a process in which the internal organization of a system increases automatically without being guided or managed by an outside source (Ashby 1947). This happens through a process of feedback and decision-making.

The law of requisite variety (Ashby 1956) which is derived from general systems theory (Umpleby 2001) also has implications to system development. There are basically two interpretations of this law. The first is that the amount of appropriate selection that can be performed is limited by the amount of information available. The second is that for appropriate regulation the variety in the regulator must be equal to or greater than the variety in the system being regulated. This means that the greater the variety within a system, the greater its ability to reduce variety in its environment through regulation (Umpleby 2001).

The law of requisite variety effects the fields of communication, information theory (Shannon 1949), and decision making (Umpleby 2001). It defines the relationship

between communication and decision making, and demonstrates the quantitative relationship between the two (Umpleby 2001).

The problems with IS development methodologies are similar in nature to the problems that von Bertalanffy and his colleagues were trying to address with general systems theory. General systems theory, "...aims to point out similarities in the theoretical constructions of different disciplines, where they exist, and to develop theoretical models having applicability to at least two different fields of study" (Boulding 1956). If the hundreds of IS development methodologies (Fitzgerald et al. 2003) are considered as "different disciplines" within the spectrum of IS development methodologies, then perhaps general systems theory holds the key to their unification.

The next step in the Wall et al. flowchart is to identify the meta-requirements of the artifact. Meta requirements describe the class of goals to which the (design) theory applies (Walls et al. 2004). The ultimate meta-requirements are to produce a reliable system within a reasonable cost range and within a reasonable time period that meets the requirements for which the system is being constructed. However, in order to proceed to the next level of design the focus of the meta-requirements must be sharpened for this particular research. A clearer definition would be: "The meta-requirements of the artifact are to provide a means by which the best possible IS development methodology (or fragment thereof) is utilized at the best possible time, based on the circumstances, context, constraints of the project".

The next step in the Walls et al. formula is meta-design. Meta-design describes a class of artifacts hypothesized to meet the meta-requirements (Walls et al. 2004). There are hundreds of software development methodologies (Fitzgerald et al. 2003). However,

while these methodologies address the more generic nature of goal one above, none meets the requirements of goal number two. Goal number two implies that the artifact(s) hypothesized to meet the meta-requirements must include the ability to select a range of methodologies or methodology fragments. Therefore one methodology cannot meet this goal. The class of artifacts hypothesized to meet the meta-requirements would be stated as: "An artifact that provides the means by which to select the best possible IS development methodology (or fragment thereof) at the best possible time, based on the context, contingencies, and constraints of the project".

Guideline # 3: Creation of an innovative, purposeful artifact

The problem has been defined, as well as a theory upon which a solution can be based, a class of goals to be met (the meta-requirements), and a class of artifacts hypothesized to meet those goals (the meta-design). It is now time to use those concepts in the creation of the artifact. That process begins by applying general systems theory.

An IS development methodology is a "system", (i.e., complexes of elements standing in interaction (von Bertalanffy 1969)), that is used to develop information systems. At first observation these systems represented by the various methodologies appear to have only a tangential relationship. However upon closer examination the isomorphic characteristics can be teased out.

IS development is essentially a problem solving process (DeFranco-Tommarello and Deek 2002, Highsmith 2000). Which means that all of the IS development methodologies are essentially problem solving systems. General systems theory tells us that a system is complexes of elements standing in interaction (von Bertalanffy 1969).

A generic problem solving system as expressed in Figure 3.2 would include the following elements:

1. Problems - the difference between a goal state and the current state of the system (Hevner et al. 2004).

2. Problem Solving Processes - the tools, procedures, processes, etc. that are used to do the following (Deek et al. 1999):

   a. Define and understand problems.

   b. Plan solutions to problems.

   c. Implement solutions.

   d. Verify and present the results.

3. Solutions - The answer to or disposition of a problem (American Heritage Dictionary 2000).

4. Feedback – Where part of the output is monitored back, as information on the preliminary outcome of the response, into the input (von Bertalanffy 1969).

5. Environment - defines the contingencies, constraints, rules, laws, etc. of the organization, people, technology, etc. Simon discusses an inner environment, an outer environment, and the interface between the two that meets certain desired goals (Simon 1996). The outer environment consists of external forces and effects that act on the artifact. The inner environment is the set of components that make up the artifact and their relationships (i.e., the organization) of the artifact.

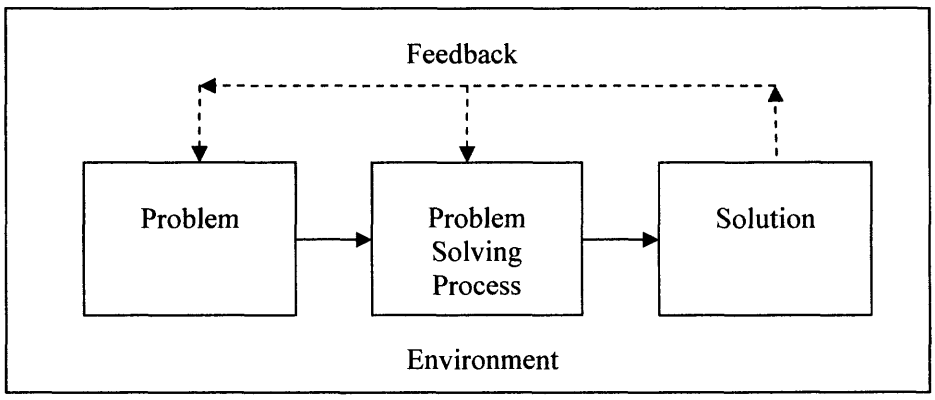


Figure 3.2 A generic problem solving system.

The problem solving systems (i.e., methodologies) employed in the IS development process are obviously more complex in nature than the system depicted in Figure 3.2. Figure 3.2 has some critical elements and contexts missing. Perhaps the most glaring component that is missing is people. IS development is a collaborative process. Therefore, IS development methodologies can be characterized as collaborative problem solving systems.

A second major component that is missing from the generic model is the nature and order of the problems. The problems in a typical IS development project are actually a series of problems and tasks (Ginat 2002) that have a hierarchical order. Hierarchy (i.e., a collection of parts with ordered asymmetric relationships inside a whole (Ahl and Allen 1996) is a concept that is central to general systems theory (von Bertalanffy 1969, Simon 1973). Hierarchical problem solving involves using intermediate states as intermediate goals in solving problems (Newell and Simon 1972). In other words, solving problems in incremental steps.

IS development methodologies can now be characterized as collaborative, hierarchical, incremental, problem solving systems. These systems have several general system theory characteristics. They are open systems that interact with their outer (Simon

1996) environment, which means that they have the propensity for negative entropy thorough a process of continuous feedback (von Bertalanffy 1969). They also demonstrate the spectrum ranging from organized simplicity (i.e., complexity involving a small number of components that interact deterministically, Weinberg 1975) to disorganized complexity (i.e., systems with many components and a considerable degree of randomness in their behaviors and interactions, Weinberg 1975). Finally, these systems all have a "system state" (Kuhn 1974), which represents the current condition of system variables (such as the current number of open, unsolved problems in the system).

Guideline #4: The artifact must be rigorously defined, formally presented, coherent, and internally consistent.

The contribution of this research will be a model artifact. Model artifacts are used to abstract and represent phenomena (March and Smith 1995, Hevner et al. 2004). The model will be called a "Collaborative, Hierarchical, and Incremental Problem-Solving" model (CHIPS). Like general systems theory, the intent of CHIPS is to provide a unification model across the disciplines of its domain. In the case of the CHIPS model, the "disciplines" refer to the plethora of system development methodologies and approaches. CHIPS is meant to model the elements of many different system development projects and to be inclusive of many approaches and methodologies. Also like general systems theory, the CHIPS model seeks to unify the disciplines, not by breaking the disciplines down, but by discovering the concepts that are isomorphic across the disciplines (von Bertalanffy 1969).

In order to properly define the CHIPS "model" the CHIPS "constructs" will first be defined. Constructs are the basic language of concepts from which phenomena can be characterized (March and Smith 1995). These concepts can then be combined into higher

order constructions (i.e., models) used to describe tasks, situations, or artifacts (March and Smith 1995).

The constructs of the CHIPS model include five classes of elements that are common across all system development projects. The five classes are: problems, solutions, people, problem solving mechanisms, and environmental concerns. Figure 3.3 and Figure 3.4 diagrams these concepts and their relationships.

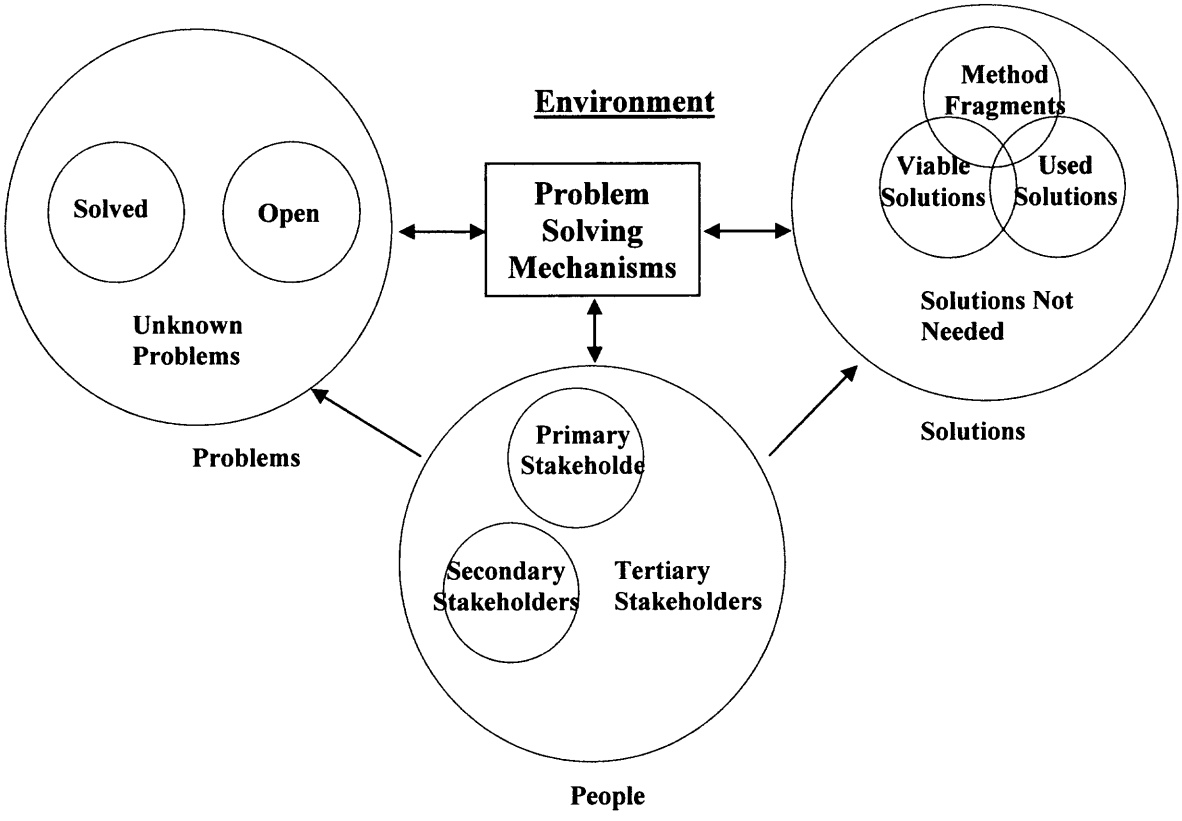


Figure 3.3 – Constructs of a collaborative hierarchical incremental problem solving model.

Defined in depth, the five constructs would be:

1. Problems - CHIPS assumes that all IS development projects are made of a series of tasks that must be completed and problems that must be solved in order to complete the project (Ginat 2002). A task can be defined as a function to be performed or an objective (American Heritage Dictionary 2000). A problem can be defined as the difference between a goal state and the current state of the system (Hevner et al. 2004). For this model, the two terms are synonymous and are thus both defined as a problem. The problems can fall into one of three sub-classes:

   a. Solved – Problems that are known and solved.

   b. Open – Problems that are known but have not yet been completely solved. In accordance with the concept of hierarchic order prevalent in general systems theory (von Bertalanffy 1969), many of the problems in a system development project are hierarchical in nature. A hierarchy is a collection of parts with ordered asymmetric relationships inside a whole (Ahl and Allen 1996). A problem may be made up of one or several subordinate problems. To successfully solve a problem, the subordinate problems must also be solved.

   c. Unknown – Problems that the developers do not know about yet, but assume will appear.

2. Solutions – A solution is the answer to or disposition of a problem (American Heritage Dictionary 2000). It can be in the form of an action, a methodology fragment, a tool, a process, a product, etc. The solutions can fall into one of four categories:

   a. Used Solutions – Solutions that have already been used in solving a project problem.

   b. Viable solutions - Solutions that can potentially be used to solve open problems for this project. This sub-class overlaps in the diagram with the used solutions class because potential viable solutions can come from the sub-class of solutions that have already been used to solve a problem.

   c. Methodology fragments - coherent pieces of IS development methods (Brinkkemper 1996). Method fragments can be divided into product fragments and process fragments. Product fragments are deliverables, diagrams, tables, models, documentation, etc. and process fragments are models of the development process (high level project strategies and detailed procedures that support specific techniques) (Brinkkemper 1996). The methodology fragments sub-class overlaps both the used solutions sub-class and the viable solutions sub-class because methodology fragments could have been used as a solution to a problem or could be a viable solution to an open problem.

d. Solutions Not Needed – these are the solutions that do not apply to this project.

3. People. – These are the stakeholders (i.e., a person or organization that has a legitimate interest in a project or entity) of the development project There are three classes of stakeholders:

   a. Primary Stakeholders – these are the people that have a direct interest in the project such as users, developers, consultants, vendors, etc. Primary stakeholders can be either in or outside of the organization in which the project is being developed.

   b. Secondary Stakeholders – these are people who have a less direct interest in the project. For instance, the manager of the department where the user of the system resides or the CEO of the organization.

   c. Tertiary Stakeholders – These are people with a very small interest in the project. This could potentially include everyone in the world. For instance, if the project were developed using an "open source" approach developers with a very minute interest in the project could be involved in the development of the project.

4. Problem Solving Mechanisms – These are the tools, procedures, processes, etc. that are used to do the following: (Deek et al. 1999)

   a. Define and understand problems

   b. Plan solutions to problems

   c. Implement solutions

   d. Verify and present the results

5. Environment - In accordance with general systems theory, the CHIPS model operates within an environment. The environment defines the circumstances, constraints, rules, laws, etc. of the organization, people, technology, etc. Simon discusses an inner environment, an outer environment, and the interface between the two that meets certain desired goals (Simon 1996). The outer environment consists of external forces and effects that act on the artifact. The inner environment is the set of components that make up the artifact and their relationships (i.e., the organization) of the artifact. The behavior of the artifact is constrained by both its organization and its outer environment. As such, activity defined through the CHIPS model must be in accordance with the laws of its environment.

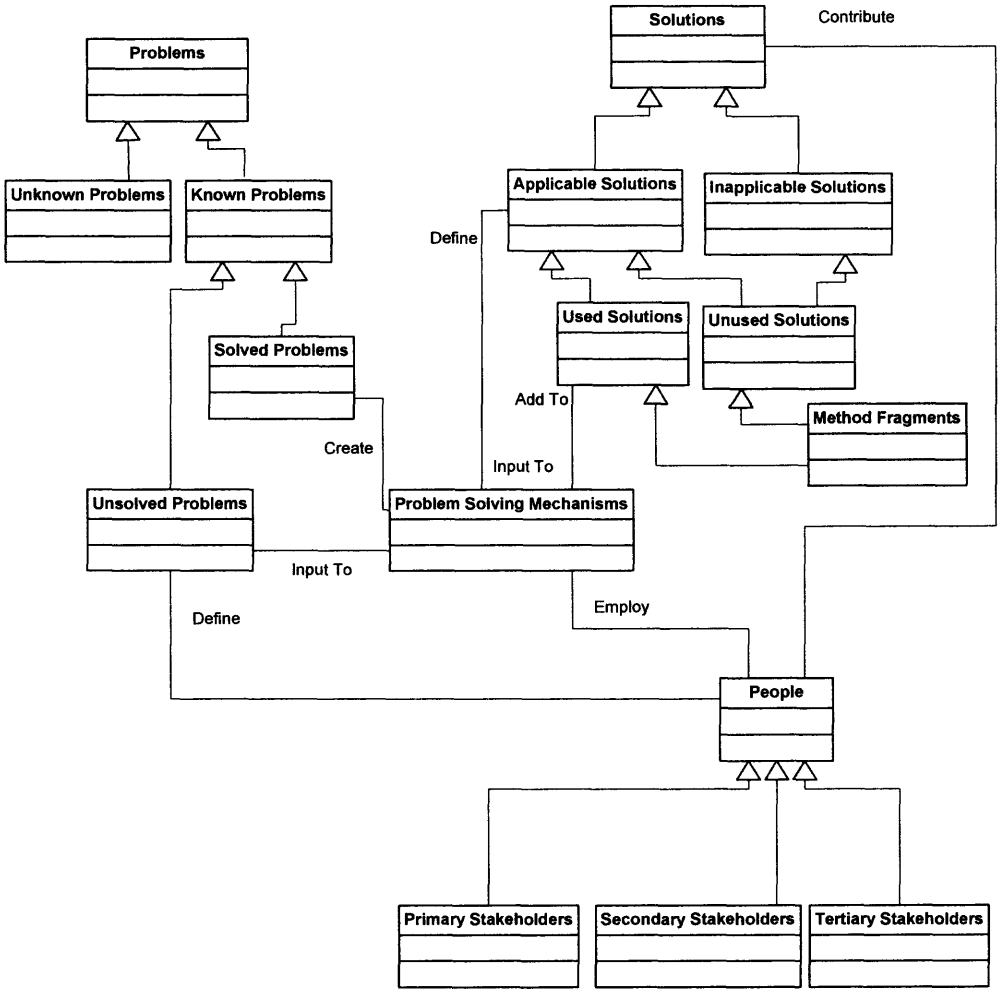


Figure 3.4 UML class diagram of the CHIPS model.

There is also an "implied" construct of the CHIPS model. That is the construct of collaboration (i.e., to work together, especially in a joint intellectual effort (American Heritage Dictionary 2000)). Whenever there are two or more stakeholders in a project, collaboration on some level would be a requirement. In the extreme case (for instance where there is one person who is both the user and the developer of the IS) then collaboration would not be a requirement of CHIPS. This, however, is also contingent on

the nature and structure of the problem. For instance programmers should be empowered to make low level programming decisions on their own.

Collaboration requires coordination (i.e., harmonious adjustment or interaction (American Heritage Dictionary 2000)) efforts at varying levels among the participants. There have been several models introduced in the literature (Daft and Lengel 1986, Allen and Hauptman 1987, Adler 1995), which discuss the organizational structure of and factors that effect coordination efforts. CHIPS is not designed to explicitly address these factors. However, CHIPS could incorporate the various models. For instance, Aldler suggests a framework that includes twelve types of coordination mechanisms built on a matrix of four approaches and three temporal phases (Adler 1995). Any of these coordination mechanisms could be employed as the project progresses through its lifecycle.

Using these constructs, the CHIPS model can now be defined. The CHIPS model is meant to represent an incremental and iterative process. It is incremental in that the number of solved problems increases in small steps as the project progresses. It is iterative in that the current state of the system is continuously redefined and used to identify problems, which are then solved and used to prescribe an action. Feedback from the previous cycle(s) becomes a significant input into the definition of the current system state.

The CHIPS model has the following phases, (as demonstrated in Figures 3.5 and 3.6), that are repeated in an iterative manner throughout the life of the project:

1. Describe:
    a. Define the current state of the project by:
        i. Analyzing the current environment.
        ii. Analyzing feedback from the previous iteration.
        iii. Identifying open problems (recall that problems are defined as problems that need to be solved and tasks that need to be completed).
    b. Decompose problems into sub-problems.
    c. Prioritize the open problems.
2. Problem Solve:
    a. Choose the highest priority problem.
    b. Apply a problem solving mechanism.
    c. Choose a solution from the knowledge base of the project (fragments from other methodologies besides the base methodologies can also serve as solutions).
3. Prescribe:
    a. Prescribe the next course of action for the project.
    b. Record the problem as close/solved (if a solution was found).
    c. Escalate the problem to a higher level of abstraction if no problem solution was found.
    d. End the project if an appropriate end point has been met.
    e. Update the knowledge base with the solution/action.

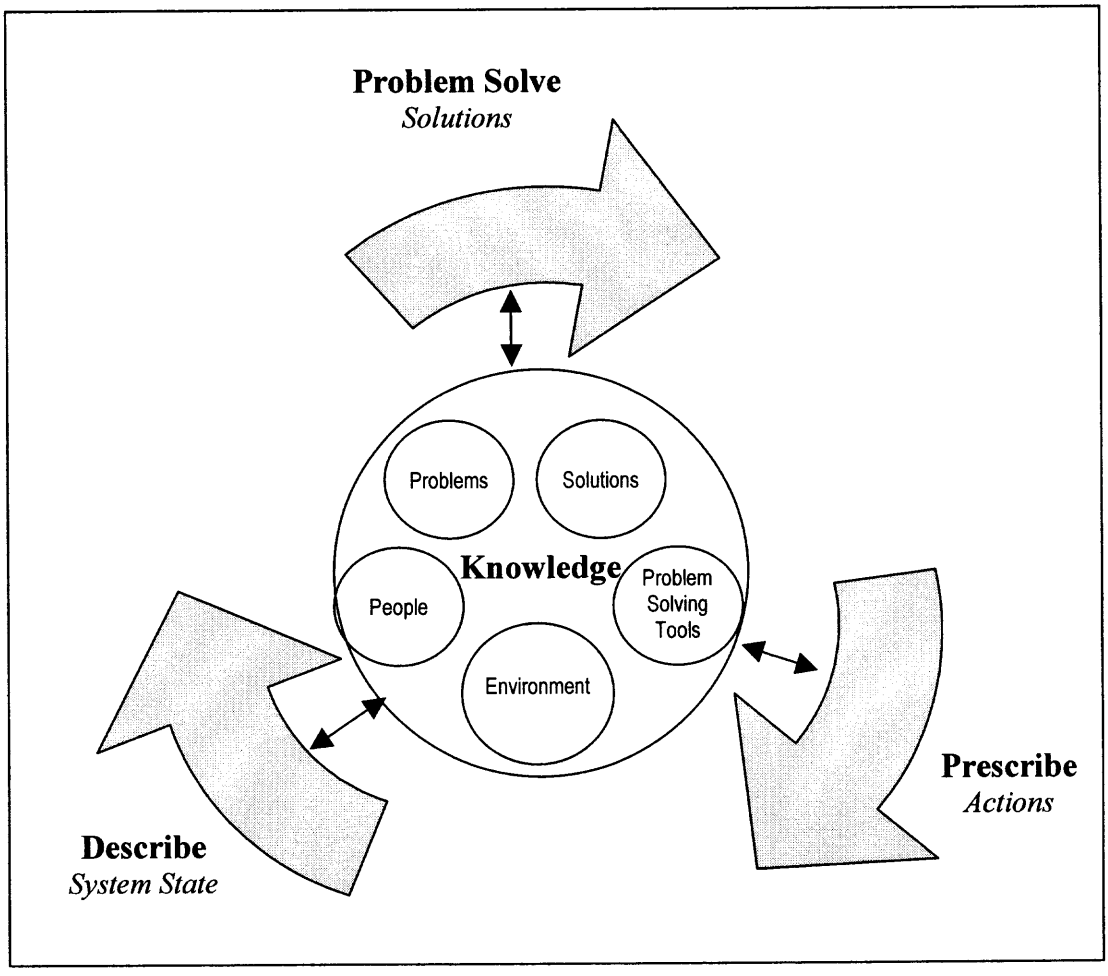


Figure 3.5 A collaborative hierarchical incremental problem solving model.

The "describe" phase is used to understand the current state of the project. It is a knowledge producing activity (March and Smith 1995). It includes analyzing the current environment and identifying circumstances that have changed since the last definition phase, analyzing feedback that was obtained from the previous iteration, analyzing and parsing the list of problems still open at the conclusion of the cycle, and adding to the list any new problems that can be identified. The list of open problems is then broken down into sub-problems, which are then prioritized.

During the describe phase, many general systems theory concepts can be applied. For instance, feedback, where part of the output is monitored back, as information on the

preliminary outcome of the response, into the input (von Bertalannfy 1969) is used to determine the outcome of previous iterations. The law of requisite variety (Ashby 1956) tells us that we are limited in our choices to the amount of information that is available. The describe phase is an information gathering phase intended to increase the amount of information and thus increase the number of choices available to system developers.

The "problem solve" phase is used to solve the highest priority problem in the list of open problems. If the problem is something simple, for instance a task that needs to be completed, then it can immediately pass to the next phase. However, if the problem is complex, then a problem–solving technique must be applied (i.e., brainstorming, Polya's method, etc) in order to collaboratively find a solution to the problem. The solution to the problem may be a methodology fragment. For instance, it may be determined that the best solution at this phase would be to build a prototype or to create an ER diagram.

The final phase is to "prescribe". This is a knowledge using activity (March and Smith 1995). Using the knowledge gained during the previous two phases the next course of action is prescribed. If a solution is found to the open problem, the problem that is solved by completing the action is now marked as a solved problem and the solution is recorded in the knowledge base. If a solution is not found, an appropriate course of action is prescribed, for example, to escalate the problem to a higher level of abstraction. Also, if it has been determined through the previous two phases that the project is at appropriate ending point, then the action prescribed may be to end the project.

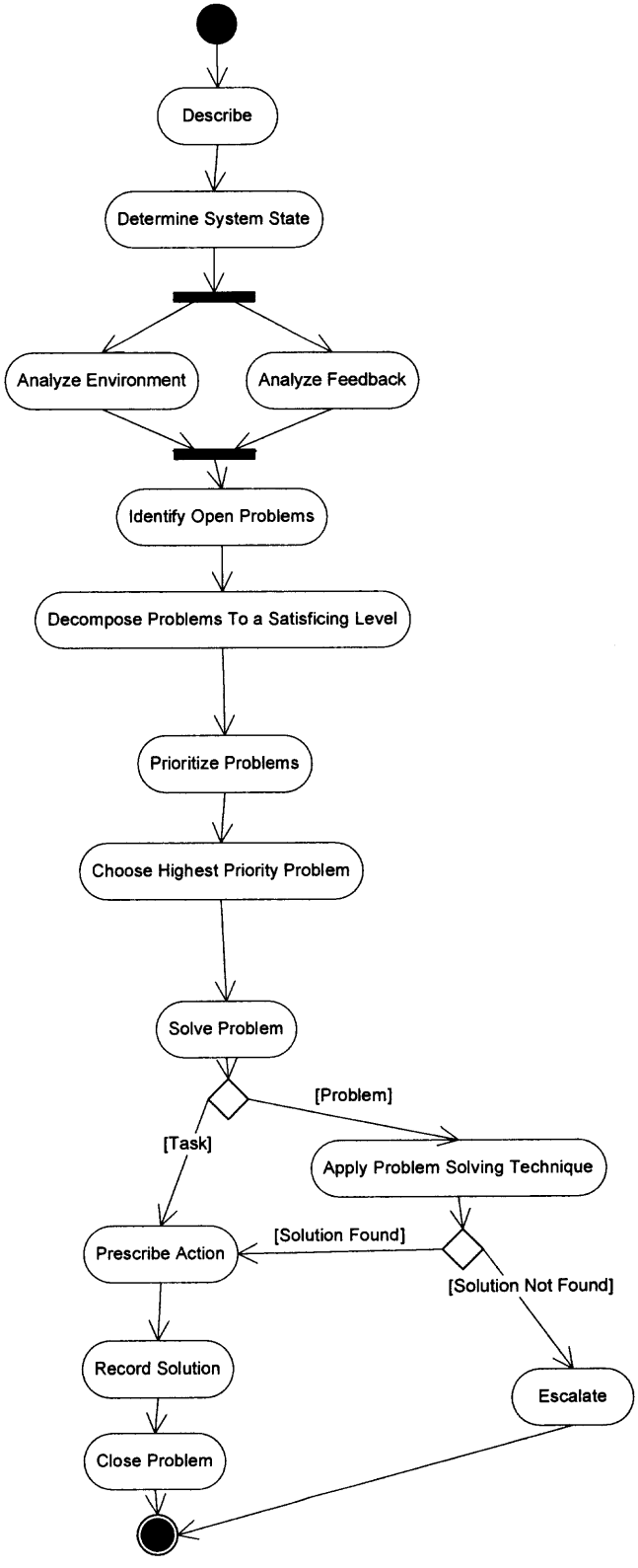


Figure 3.6 A UML activity diagram of the CHIPS model.

CHIPS attempts to abstract methodologies to a common level. It assumes that all methodologies involve:

1. People working together collaboratively
2. A set of unsolved problems and tasks
3. An environment which defines the parameters and constraints of the project
4. An algorithm for decomposing and prioritizing the unsolved problems and tasks.
5. A set of processes for solving the problems and completing the tasks in incremental steps.

By abstracting software development projects to the level and components suggested by CHIPS, the focus is placed on solving the problems and completing the tasks inherent to the project, rather than the employment of a methodology. This allows the developers to concentrate on improving and completing the core processes.

Methodology fragments can easily be selected, arranged, and inserted into the process. This allows the process to become as prescriptive or as adaptive as necessary and thus allow the project to meet the visibility requirements of management, yet adapt to unforeseen change.

The CHIPS model has several advantages:

1. Prescriptive/Evolutionary – CHIPS can be used to either prescribe a course of action or adapt to new issues that arise. It forces developers to identify problems (i.e., tasks), solutions, what people will participate, what problem solving mechanisms will be used for the project, and the current environment. IS development methodologies can be used to prescribe a course of action for the project and pre-defined problem solving mechanisms can be used to solve new problems as they arise.

2. Simple/Complex – it can be made as simple or complex as needed, dependent on the project.

3. Linear/Non-Linear – it can structure the tasks/problems in a linear fashion (such as in a typical "waterfall" approach) or in a non-linear fashion.

4. Dynamic and learning – the solutions class and the problem solving class are dynamic, expanding, learning, and remembering. In the future a CHIPS tool will have search features to facilitate matching solutions to tasks/problems.

5. It can encapsulate many development project and many sets of development methodologies.

In order to properly define the CHIPS model, a methodology (algorithms and practices), will be presented that fits the paradigm. It must be pointed out that the principle of equifinality (von Bertalanffy 1969) holds true in the model. Equifinality is a condition in which different initial conditions lead to similar effects or in which different courses of action lead to similar results. Application of this principle suggests that there are multiple methodologies and instantiations that would fit the CHIPS model and still produce the desired result.

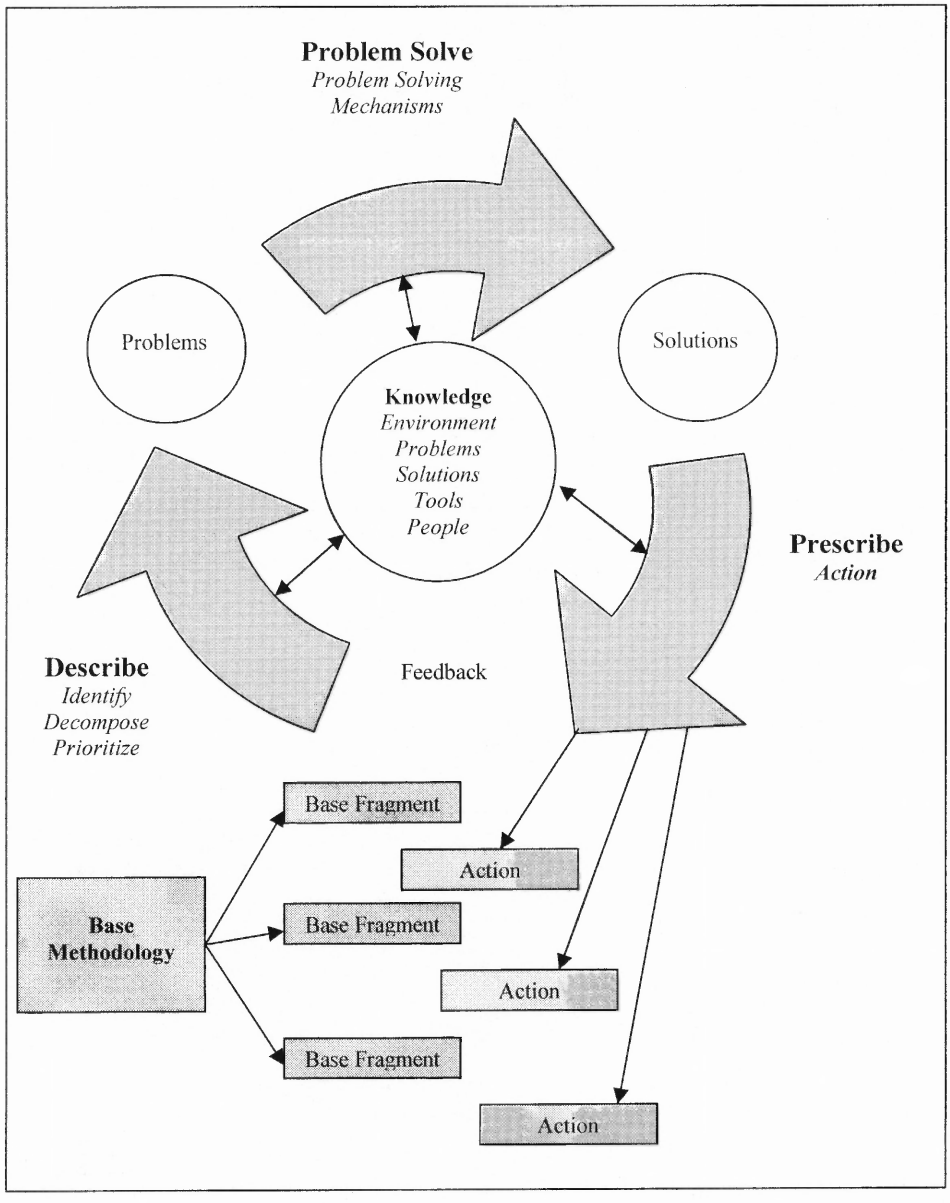


Figure 3.7 A collaborative hierarchical incremental problem solving methodology.

The methodology is illustrated in Figure 3.7. Generally, it has a cyclical nature that involves refinement (to improve by eliminating unnecessary elements), and tailoring (to fit by exacting adjustments). The previously described theoretical foundations, model and ME method fragment concept are used to positively adapt both generic and custom SDLC and process methodologies. The method oscillates between a prescriptive and descriptive analysis using the output or feedback (as is consistent with General Systems Theory) as input to the other approach.

The process begins by selecting a base methodology with core competencies, (i.e., The set of the most strategically significant and value-creating skills in any organized system or person), that most closely match the contingencies of the project and the organization. Several key factors contribute to this selection process, for instance, the knowledge and background of the developers.

Once the base methodology has been selected, the next step is to extract the fragments from it that will serve as a skeleton methodology for the project. These fragments are arranged in a in a temporal fashion, with intentional gaps left in the prescribed process. This is represented by the orange components in Figure 3.7.

The methodology now progresses into a cyclical (iterative) process. The process has the following phases (as defined by the previous model):

1. Describe:
    a. Define the current state of the project.
    b. Identify open problems.
    c. Dissect problems into sub-problems.
    d. Prioritize the open problems.
2. Problem Solve:
    a. Choose the highest priority problem.
    b. Apply a problem solving mechanism.
    c. Choose a solution from the knowledge base of the project (fragments from other methodologies besides the base methodologies can also serve as solutions).
3. Prescribe:
    a. Prescribe the next course of action for the project.
    b. Record the problem as close/solved.
    c. Update the knowledge base with the solution/action.

The methodology continues to follow this cycle throughout the course of the project. The base methodology fragments that were initially extracted as the skeleton methodology serve as anchor points which keep the project grounded. The prescribed actions must be collated within the fragments of the base methodology that were initially prescribed. The methodology can continue to be employed throughout the lifecycle of the project, even after the project as progressed into the maintenance phase.

An instantiation (March and Smith 1995) of CHIPS can be demonstrated by taking a popular system development methodology and showing how it can be improved. One of the most popular methodologies is RUP. There have been several articles written on the deficiencies of RUP. Chief deficiencies are:

1. It is complex and complicated and can be difficult to learn.
2. It is expensive to purchase.
3. Training, budgeting, and process measurement are not explicit.
4. It is still a phase oriented life cycle model (i.e., prescriptive).
5. It does not include tools to handle complexity (i.e., It lacks hierarchy, recursion, and orthogonality).
6. It does not provide adequate management support.
7. It ignores the role of feedback of users in the project.
8. It does not handle re-use.
9. It does not cover the maintenance phase.

CHIPS can be used to augment RUP and correct some of these deficiencies. There are two ways that this can happen. Another methodology can be chosen as the base methodology, and then the process can be enhanced with fragments from RUP, or RUP could be the base and enhanced with fragments from other methodologies and/or solutions to problems outside the scope of known methodologies.

The integration of a base methodology with RUP via the CHIPS model can be demonstrated by analyzing a fictitious project. Let's say that the fictitious development project is to create a new information system to for a government agency. Using the CHIPS model, the project is initiated by executing the first define phase. The

environment is analyzed and a list of open problems is created and decomposed into sub-problems. The list is then prioritized. The highest priority problem right off the bat is to select a base system development methodology for the project.

The CHIPS model now mandates that the developers execute the problem-solving phase. The project is one that requires a very high level of visibility and thus a high level of management support (because it is being paid for by taxpayer money). Due to its inadequacies at providing management support, RUP is not chosen as the base methodology for the project. A more rigid, visible methodology such as the waterfall methodology is chosen as the base methodology. However, there is a strong chance that the system requirements will undergo several changes throughout the life of the project, so the developers would like to integrate some RUP features into the project. Furthermore, the developers are trained in OO development and would like to capitalize on the enhanced modeling features inherent to RUP.

The first CHIPS cycle ends by prescribing the traditional SDLC as the base methodology for the project, with the understanding that the process will be augmented with RUP features. Figure 3.8 gives a visual idea of how the integration process takes place. Since SDLC is the base methodology, a general system development plan is laid out that includes the typical phases of the waterfall methodology (as illustrated by the orange rectangles in the figure). The initial SDLC phase (system investigation) is then completed using only the method fragments inherent to the waterfall approach. Throughout this phase and all subsequent others the project iterates through the define, problem solve, and prescribe activities of the CHIPS process.

During the analysis phase, data collection is performed using the traditional methods (i.e., interviews, surveys, questionnaires, etc.). However, instead of using traditional modeling techniques (i.e., data flow diagrams, flowcharts, E-R diagrams, etc.) to abstract the data collected, RUP modeling techniques are used. For instance, class diagrams, state diagrams, and activity diagrams would be used to demonstrate the system.

The project eventually progresses to the design phase. Both the logical and physical designs are accomplished through the employment of RUP techniques to model the new system's features and the development team's understandings and agreements. Using RUP, the team designs key objects and classes of objects in the new system. This process involves consideration of the problem domain, the operating environment, and the user interface. The problem domain involves the classes of objects related to solving a problem or realizing an opportunity.

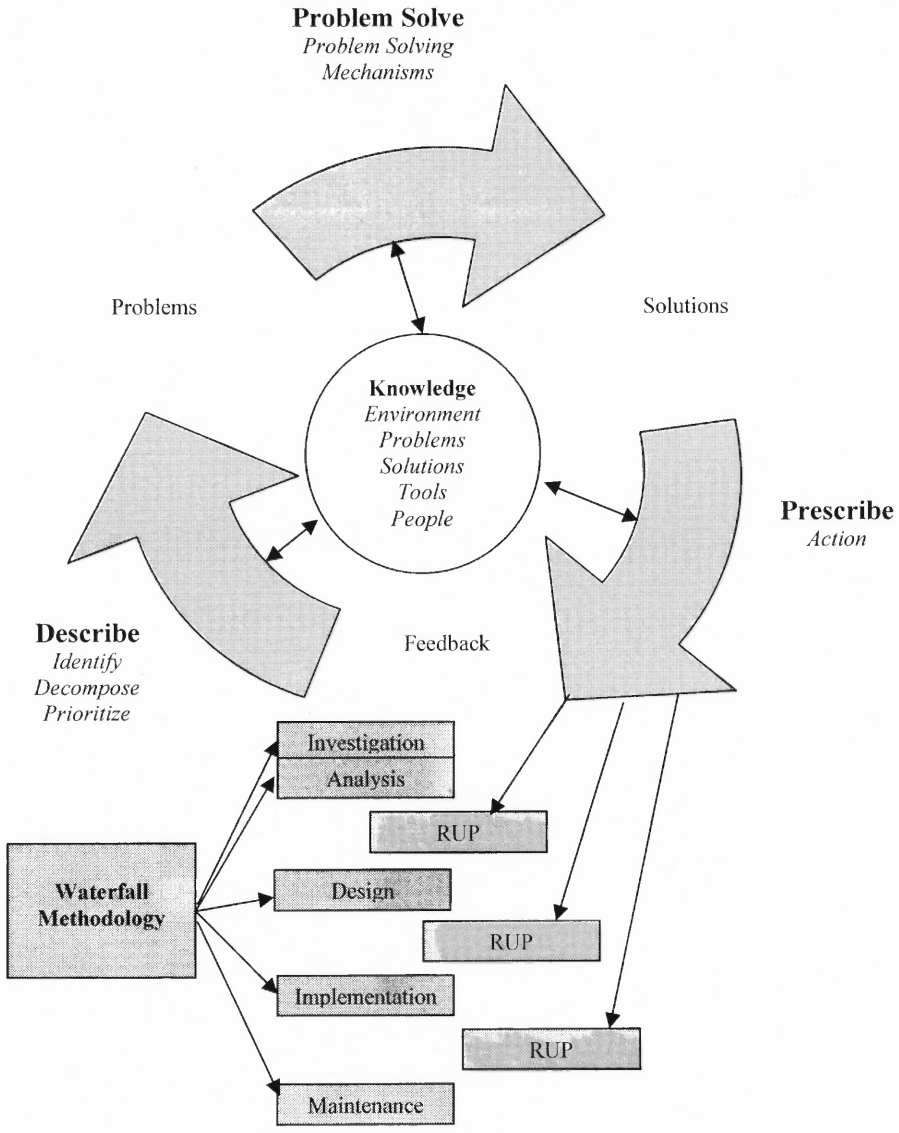


Figure 3.8 Waterfall augmented with RUP through the CHIPS model.

During the design phase, the development team also needs to consider the sequence of events (scenarios) that must happen for the system to function correctly. This sequence of events can be diagrammed in a sequence diagram. The RUP concepts of

iteration and incrementation can also be implemented as the system design is refined through continuously adding more detail to the UML diagrams.

Eventually the project progresses to the implementation phase. Using RUP, the object model begun during analysis and completed during design is turned into a set of interacting objects in a system. Object-oriented programming languages are used to create classes of objects that correspond to the models (or CASE tools such as Rational Rose are employed).

Using the RUP paradigm, the initial implementation is evaluated by users and improved. Additional objects and scenarios are added incrementally as the project iterates through this phase. Eventually a complete, tested, and approved system is available for use.

Once the system enters the maintenance phase, the project reverts back to the traditional waterfall approach, with one exception. The project continues to follow the define, problem solve, prescribe cycle of CHIPS. However, once in the maintenance phase, the frequency of these cycles would be much less.

Guideline #5: The artifact must solve an unsolved problem in an innovative way

The meta-requirements of the artifact have previously been defined as "to provide a means by which the best possible IS development methodology (or fragment thereof) is utilized at the best possible time, based on the context, contingencies, and constraints of the project". Two formalized approaches have previously been defined that are prior attempts to fulfill those meta-requirements; the contingency factors approach (where a methodology is chosen for a project based on a set of contingent factors) (Avison and Fitzgerald 2003, Fitzgerald et al. 2003) and method engineering (where methodologies

are decomposed into a repository of methodology fragments and then re-assembled into met-methodologies of optimum fragments using a special tool) (Brinkkemper 1996). There is also one more approach that is informal in nature. That is the ad hoc methodology tailoring that takes place by practitioners in industry as they adapt methodologies to match their circumstances (Fitzgerald et al. 2003). However, it is the goal of this research to present a more formalized model than this ad hoc methodology.

The inadequacies of the contingency factors approach are apparent (Fitzgerald et al. 2003). It is just not feasible or possible for all the developers in an organization to be familiar with all of the possible methodologies that would work best for a given situation (Fitzgerald et al. 2003). Plus as the contingent factors of the project change over time, so will the optimum methodology.

If method engineering is analyzed through the lens of general systems theory, it becomes apparent that it is both a reductionistic and mechanistic solution to the problem. It is reductionistic in the sense that it attempts to solve the problem by reducing the phenomenon (the methodology) to its smallest component (method fragments) and analyzing the components. It is mechanistic because it attempts to build a whole meta-methodology from the sum of its parts, with no regard for the interrelationships of those parts. It was Aristotle who first stated, "The whole is more than the sum of the parts".

CHIPS is a holistic, anti-reductionistic, anti-mechanistic approach. It seeks to integrate (and thus fulfill the meta-requirements of the artifact) by identifying and capitalizing on the isomorphic characteristics of the IS development methodologies. In an open system negative entropy occurs through a process of decision making and feedback (von Bertalanffy 1969). Huber showed us that the definition of problem solving

is the decision making process augmented with implementation and feedback. Therefore it stands to reason that a model that focuses on problem solving would be one that leads to higher levels of order over time.

Guideline # 6: Thorough evaluation of the artifact

In accordance with guideline three (Hevner et al. 2004) the CHIPS model must have a thorough scientific evaluation. Chapter 4 of this document presents details of the research methods that were used to evaluate the model. In summary, a lab experiment was conducted to test the theory that CHIPS will enhance the system development process.

Guideline #7: The results of the research must be communicated effectively.

The purpose of this document and of this research is to effectively communicate the results of the research

3.3 Benefit of the contribution

The contribution of this project is to create an IS development model that provides guidance and:

1. Enhances existing IS development methodologies.
2. Will provide an explicit collaborative problem-solving environment that can be used to adapt the process to conditions that arise during the system development process.
3. Makes traditional methodologies more adaptive to change.
4. Makes agile methodologies more visible.
5. Will provide a framework for future research for academics.

Given the statement of the contribution of this research to the community at large, the question then arises as to who will benefit from this research. The benefactors can be listed as:

1. Ultimately the end-users of computer software, which could be potentially a large portion of the population of the world. This model is being proposed as potential solution for building better software, and therefore would benefit anyone that uses an IS developed by its process.
2. IS development practitioners – These are the people that are out developing information systems in the real world. If this model can provide a better means by which they can develop systems, then they can do their jobs more effectively and efficiently.
3. The academic community – CHIPS is proposed as a model of inclusion of all the specific research areas that are being undertaken in the realm of IS development methodologies and collaborative, software development problem solving. Hopefully, this model can be used as a framework for bridging the gap between these various "islands" of research. Also, a goal of this model is to find a common ground upon which academics and practitioners can stand in their pursuit of better IS development methodologies.

3.4 Research Questions

Given the model, the research questions can now be stated as:

Q1: Will developers be more satisfied with the system development process when using CHIPS than when not using CHIPS?

Q2: Will developers be more satisfied with the finished system developed when using CHIPS than those developed when not using CHIPS?

Q3: Will developers be more satisfied with the problem solving capabilities of their development group when using CHIPS than those groups who do not use CHIPS?

Q4: Will expert judges rate the finished design of systems developed using CHIPS better than the design of systems developed not using CHIPS?

CHAPTER 4
RESEARCH METHODOLOGY

4.1 Experiment Basis

In order to evaluate the CHIPS IS development model a laboratory experiment was conducted. The CHIPS experiment was based on a prior experiment, (Alavi 1984), in which the prototyping approach to system development was evaluated by having a control group develop a system using a traditional waterfall methodology and an experiment group develop the same system using the prototyping methodology. The results obtained by the two groups were then compared and analyzed.

In contrast to prototyping, which is a specific IS development methodology, CHIPS is a methodology-tailoring approach used to adjust a methodology to the circumstances of the project (which could result in the combination of *several* IS development methodologies). Therefore, the experiment had to be adjusted accordingly.

There are currently three known method tailoring approaches; Method Engineering, Contingency Factors, and Ad-hoc. Prior research has already shown the advantages of method tailoring over individual methodologies (Cockburn 2002, Fitzgerald et al. 2003). In order to evaluate CHIPS, it had to be compared to one of the three known method tailoring approaches and not to a specific IS development methodology (as in the case of the Alavi experiment). Given the constraints of a lab experiment, the logical choice was to compare CHIPS to ad-hoc method tailoring.

The original Alavi experiment used students as subjects. The students were selected from either a pool of MBA students or a pool of MS students majoring in IS. The MBA students were used to simulate real world business IS users and the IS students

were used to simulate IS developers. The students were split into development teams with each team having 3 or 4 users and 3 or 4 developers. The teams were then asked to develop a specific IS. Some of the teams used the traditional SDLC method to develop the system and some of the teams used the prototyping methodology.

Once the systems had been developed, the students were asked to complete Likert scale questionnaires. The questionnaires measured three things:

1. User evaluation and satisfaction with the IS.
2. User and designer perceptions of the design process.
3. The extent that the developed IS would be used to solve a business problem.

4.2 Experiment Overview

The experiment to evaluate CHIPS was a 2 x 2 between subjects factorial design. Experimental designs in which every level of every variable is paired with every level of every other variable are called factorial designs. A 2 x 2 means that there will be two independent (or manipulated) variables each of which can have two levels or treatments. A between subjects experiment is one in which the subjects are only exposed to one treatment.

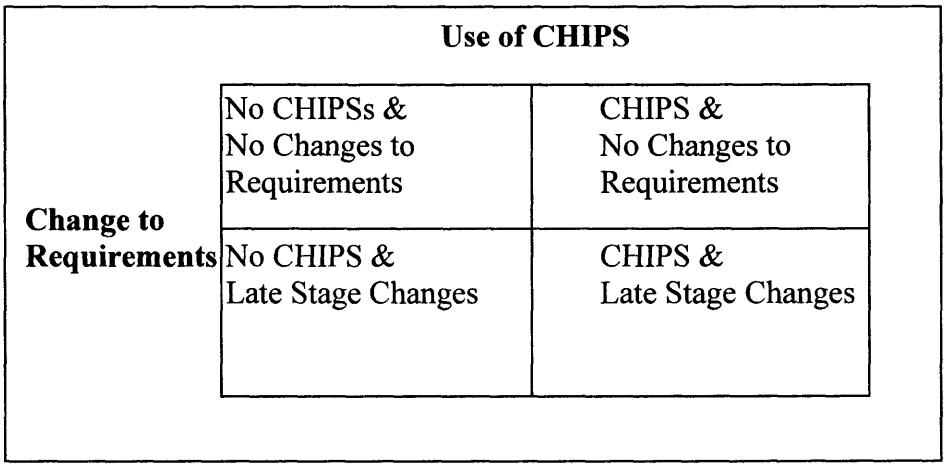


Figure 4.1 A 2 x 2 experiment to evaluate CHIPS

Figure 4.1 demonstrates the design of the experiment. The two independent variables that were manipulated were the use of CHIPS (used vs. using the ad-hoc approach) and the introduction of requirements changes in late stages of the development process (changes vs. no changes). This design allowed the subjects to be assigned to one of four treatments:

1. Do not use CHIPS (Ad-hoc) and have no changes to requirements.
2. Do not use CHIPS (Ad-hoc) and have late stage requirements changes.
3. Use CHIPS and have no changes to requirements.
4. Use of CHIPS and have late stage requirements changes.

Steps were taken to ensure that the CHIPS subjects would not have an advantage over the ad-hoc subjects. Subjects in all treatments were given the same introduction and the same level of instruction covering information systems development techniques and their commonality with the problem solving process, including open problems, problem

analysis, and solution design. All subjects were also introduced to three commonly used problem solving methods (Polya's method, brainstorming, and SWOT analysis).

The CHIPS (experiment) group was asked to use an instrument that explicitly prompted the participants to adhere to the information systems development techniques and problem solving methods introduced earlier to all subjects in both groups.

The experiment measured four dependent variables:

1. Subjects' satisfaction with the development process.
2. Subjects' satisfaction with the finished design.
3. Subjects' satisfaction with their group's problem solving capabilities.
4. Expert judges' rating of finished design.

Subjects' satisfaction with the development process, the finished design, and their group's problem solving capabilities was measured using questionnaires based upon the instruments used in the original Alavi experiment. These original instruments utilized 5 point Likert scales whose items had been statistically validated (Alavi 1984). Three expert judges were used to rate the final product created by the subjects.

The experiment followed the same procedure as the Alavi (1984) experiment in that students were used as subjects. The students were divided into groups of development teams (three or four students per team). Teams were then randomly assigned to one of the four treatments.

The task assigned to all teams was to design an e-commerce web site. The task involved only design; no program coding or implementation took place. All of the groups were given the same initial requirements of the web site. They were asked to design the web site and describe their design using graphical representations of the web

pages, documentation, diagrams, and tables. Before they began, the subjects were given appropriate training based on the treatment to which they are assigned. The subjects were then given three weeks to complete the design. Two of the treatments were given a significant change to the requirements after the first week.

Once the teams have completed the project, the subjects were given a post-test questionnaire to measure the dependent variables. Three expert judges were used to review and rate each of the finished designs. Appropriate statistical tests were then conducted to validate the instruments used and to analyze the data retrieved.

4.3 The Hypotheses

There were twelve hypotheses, stated as follows:

- H1a - Developers will be more satisfied with the finished system design when it is developed using CHIPS than when it is developed not using CHIPS.
- H1b - When system requirements remain constant throughout the development process, developers will be more satisfied with the finished system design when it is developed using CHIPS than when it is developed not using CHIPS.
- H1c - When system requirements change late in the development process, developers will be more satisfied with the finished system design when it is developed using CHIPS than when it is developed not using CHIPS.
- H2a - Developers will be more satisfied with their problem solving capabilities developing a system using CHIPS than when they are not using CHIPS.
- H2b - When system requirements remain constant throughout the development process, developers will be more satisfied with their problem solving capabilities developing a system using CHIPS than when they are not using CHIPS.
- H2c - When system requirements change late in the development process, developers will be more satisfied with their problem solving capabilities developing a system using CHIPS than when they are not using CHIPS.
- H3a - Developers will be more satisfied with the development process when using CHIPS than when not using CHIPS.

- H3b - When system requirements remain constant throughout the development process, developers will be more satisfied with the development process when using CHIPS than when not using CHIPS.

- H3c - When system requirements change late in the development process, developers will be more satisfied with the development process when using CHIPS than when not using CHIPS.

- H4a – Expert judges will rate the finished system design better when developers use CHIPS to develop the system than when developers do not use CHIPS.

- H4b – Expert judges will rate the finished system design better when system requirements remain constant throughout the development process and developers use CHIPS than when system requirements remain constant throughout the development process and developers do not use CHIPS.

- H4c - Expert judges will rate the finished system design better when system requirements change late in the development process and developers use CHIPS than when system requirements change late in the development process and developers do not use CHIPS.

4.4 Experiment Details

The experiment to measure CHIPS ran over two consecutive semesters (Fall 2005 and Spring 2006) with no significant changes to the experiment design between the two tests. A total of 140 subjects participated in the experiment (40 in the fall semester and 100 in the spring semester).

The experiment drew many of its components from the original Alavi experiment (Alavi 1984). The subjects used to simulate the system developers were IS students (either graduate MS students or undergraduate students participating in their senior capstone course) and MBA students were used to simulate the actual users of the system.

The subjects were drawn from classes conducted at the New Jersey Institute of Technology. The experiment was presented as an optional assignment in the selected courses. Those not willing to participate were given an alternative assignment. At the start of the experiment, the subjects were asked to complete a consent form (Appendix A)

and a demographic questionnaire (Appendix B). The purpose of the pre-experiment demographic questionnaire was to determine the homogeneity of the subject pool and to serve as a potential source to explain any anomalies encountered.

The MBA students were assigned the role of users/judges for the experiment. Several weeks prior to the start of the experiment they were given the expert judges' task list (Appendix C). They were told that they would present themselves as business owners who wanted an e-commerce web site. They were asked to work up a rough requirements document that would be presented to the subjects. They were also asked to develop a grading rubric prior to the start of the experiment. Both of these documents are included in the subjects task list (Appendix D).

Prior to the experiment start, the subjects were formed into groups and randomly assigned to one of the four treatments:

1. Do not use CHIPS and have no changes to requirements.
2. Do not use CHIPS and have late stage requirements changes.
3. Use CHIPS and have no changes to requirements.
4. Use CHIPS and have late stage requirements changes.

The experiment was initiated through a presentation to the subjects in their classes. All subjects were told that they were participating in an experiment to measure IS development and problem solving. At the start of the presentation, they were told their group number, asked to complete the consent form and demographic questionnaire. They were then given the subject task list (Appendix D) and told that they would be acting as system developers and that each team was competing to create the best e-commerce web

site design. A brief question and answer session followed where the subjects were given clarification on their roles.

The expert judges/users were then asked to enter the room. In accordance with the expert judge task list (Appendix C) they presented themselves as business owners in need of an e-commerce web site. They went over the requirements of the web site and the criteria that they would use to rate the designs submitted by the subjects. The subjects were given a list of deliverables and due dates for the project. The subjects were allowed to ask any questions they had of the business owners and were given an email address that could be used to communicate with the business owners.

After the expert judges exited the room, all subjects were given a brief overview of three problem solving techniques; the Polya method, brainstorming, and strengths, weaknesses, opportunities, and threats (SWOT) analysis. The documentation on these techniques that was presented to the subjects is included in Appendix E. All treatment groups were exposed to the problem solving techniques so that there were would be no disparities between knowledge and awareness of problem solving techniques. The subjects were told that they were not required to use these problem solving techniques to complete their designs. This completed the experiment presentation for non-CHIPS subjects.

In a real-life situation, CHIPS would normally be implemented via a CASE tool. However, as no such tool yet exists and in order to test the CHIPS model in a laboratory experiment, a simplified tool and procedure had to be created. Both of these instruments had to be easy enough to learn and use so that their employment would not impede the developers work and skew the experiment results. However, the instruments had to be

robust enough so that they would give an accurate depiction of the CHIPS model. The solution was to create a spreadsheet that the subjects would use to organize the project into problems/tasks and to assign the tasks a priority and a collaborative responsibility. The spreadsheet also allowed the subjects to assign a problem solving technique to a problem/task and to enter comments as to how the problem was solved.

Appendix F shows the documentation that was given to the subjects. Included in the documentation were a procedure and a CHIPS worksheet. The subjects were instructed to follow the procedure and to submit a CHIPS worksheet every three days throughout the course of the experiment. Subjects were also given an instruction session where a small sample project (i.e., washing the car) was completed using the assigned CHIPS procedure and worksheet. The CHIPS subjects were given an email address to use to ask questions if they encountered any problems using the CHIPS model.

As stated earlier, all subjects were given a schedule where deliverables were due every few days. This was to prevent teams from waiting until right before the end to complete the entire project and thus ensuring that the introduction of a late stage change in the requirements, (to those teams assigned to the change treatment), would successfully simulate a real world change late in the development process. Appendix G shows the late stage change that was introduced to the subjects. In essence, the teams were asked to change the design of their web site from being a single product site, to one that could serve as a template for any e-commerce web site.

Once the subject teams completed and submitted their final projects, they were asked to complete a final post-test questionnaire. The questionnaire was completed

online via a hosted web site. The final questionnaire is included as Appendix H. The data collected and the statistical analysis of such is presented in the next chapter.

The final projects were then submitted to the blinded expert judges for grading/ranking. The projects were split into two groups, those who received the late stage change, and those that did not. This was done to ensure that like tasks were graded accordingly. Each judge was given a subset of the projects to grade at first and then eventually the projects were rotated among the judges so that each judge graded each project. The judges were not allowed to confer with each other during the grading period. This was done so that judges would not influence each other's grading. The data collected from the expert judges and the statistical analysis of such is presented in the next chapter.

CHAPTER 5
RESULTS

5.1 Pilot Experiment

The experiment was conducted in two significant phases, one in the fall of 2005 and one in the spring of 2006. Although there were no major changes in the procedures of the experiment between the two executions, the initial run in the fall of 2005 was considered the pilot. Results from the pilot are presented first.

The questionnaire was validated using Cronbach's alpha. The expert judges' reliability was validated by performing a bivariate Pearsons two tailed test of correlation and a paired samples t-test of significance. Tests of significance were then performed on the variables. The results are displayed in Tables 5.1, 5.2, and 5.3.

Table 5.1 Non-CHIPS versus CHIPS, Means and t-test Probability

	Satisfaction with Final Design	Satisfaction with Problem Solving	Satisfaction with Development Process	Expert Judges Ranking
Non-CHIPS	6.04	5.68	5.87	87.75
CHIPS	6.29	6.04	6.11	88.21
Probability	0.139	0.05	0.119	0.45

Scores measured on 7-point Likert Scales (7=High) for columns 2-4
Scores measures on scale of 1-100 for column 5
The difference between the means is statistically significant at $p<.05$

Table 5.2 Non-CHIPS/No Change Versus CHIPS/No Change, Means and t-test Probability

	Satisfaction with Final Design	Satisfaction with Problem Solving	Satisfaction with Development Process	Expert Judges Ranking
Non-CHIPS/No Change	6.00	5.58	5.75	89.50
CHIPS/No Change	6.43	6.40	6.29	83.50
Probability	0.12	0.004	0.038	0.818

Scores measured on 7-point Likert Scales (7=High) for columns 2-4

Scores measures on scale of 1-100 for column 5

The difference between the means is statistically significant at $p<.05$

Table 5.3 Non-CHIPS/Change Versus CHIPS/Change, Means and t-test Probability

	Satisfaction with Final Design	Satisfaction with Problem Solving	Satisfaction with Development Process	Expert Judges Ranking
Non-CHIPS/Change	6.10	5.80	6.00	86.00
CHIPS/Change	6.20	5.78	5.98	92.92
Probability	0.374	0.524	0.528	0.019

Scores measured on 7-point Likert Scales (7=High) for columns 2-4

Scores measures on scale of 1-100 for column 5

The difference between the means is statistically significant at $p<.05$

The results from the pilot experiment show that the following hypotheses were supported:

- H2a - Developers will be more satisfied with their problem solving capabilities developing a system using CHIPS than when they are not using CHIPS.

- H2b - When system requirements remain constant throughout the development process, developers will be more satisfied with their problem solving capabilities developing a system using CHIPS than when they are not using CHIPS.
- H3b - When system requirements remain constant throughout the development process, developers will be more satisfied with the development process when using CHIPS than when not using CHIPS.

- H4c - Expert judges will rate the finished system design better when system requirements change late in the development process and developers use CHIPS than when system requirements change late in the development process and developers do not use CHIPS.

Hypotheses H1a, H1b, H1c, H2c, H3a, H3c, H4a, and H4b were not supported. The results from the pilot experiment did identify several areas were there were statistically significant differences between the use of the CHIPS model versus not using the CHIPS model, even though the CHIPS developers generally scored higher in almost all of the areas analyzed. As for the first dependent variable, developer satisfaction with the solution, there were no significant differences between CHIPS and non-CHIPS developers. This held true regardless of whether late stage changes were introduced or not. However, the second dependent variable measured, developer satisfaction with their problem solving process did show some significant differences between CHIPS and non-CHIPS developers.

CHIPS developers, overall, were more satisfied with their problem solving process than those developers who did not use CHIPS. This would stand to reason as the CHIPS model is essentially a problem solving system. This rule also held true when

there were no late stage changes introduced to the project. However, surprisingly, when there were late stage changes introduced to the project there was no significant difference between the way CHIP and non-CHIPS developers felt about their problem solving process.

There were also some significant differences measured in the third dependent variable, developer satisfaction with the development process. In particular, developers were more satisfied with the development process when there were no late stage changes and they used CHIPS as their development model. However there was no statistically significant difference with regard to developer satisfaction with the development process for CHIPS versus non-CHIPS users overall or if there were late stage changes introduced.

The last dependent variable that was measured was the expert judges rating of the finished projects. When there were late stages introduced to the project, there was a very significant difference in CHIPS developers versus non-CHIPS developers, with CHIPS developers scoring significantly higher. This would make sense as the essence of CHIPS is to make the developers better able to adapt to change and thus provide a better product when there is change late in the development process.

The results of this initial experiment, while promising, are significant for several reasons. First, mean scores were higher for CHIPS developers than they were for non-CHIPS developers for almost all of the dependent variables. This can tentatively be used as an indicator that CHIPS has a role as a legitimate model. Second, there were areas that showed a statistically significant improvement of CHIPS developers over non-CHIPS developers. Given the qualifications of the experiment elaborated earlier, this could be an indicator that CHIPS has the potential to be a model that improves the development

process. Finally, the results of this experiment give support to the argument that CHIPS can be used to integrate several system development methodologies, not by breaking those methodologies down into method fragments, but by identifying the isomorphic characteristics of those methodologies.

5.2 Overall Results from the Experiment

Data on the use of CHIPS model was gathered using the previously described experiment over a six month period starting in November of 2005 and continuing into May 2006. This section will show the results of the data that was collected, statistical validation of the instruments used to collect the data, tests of statistical significance, and tests of correlation.

5.2.1 Validation of the Questionnaire

There were two primary instruments used to collect the data. The subjects acting as developers completed a questionnaire (appendix H) and the judges who rated the finished projects used a standardized grading sheet to compute the scores (appendix I).

Initial statistical tests were completed in order to validate the questionnaire and the expert judges grading. Cronbach's alpha was used to validate the questionnaire. Cronbach's alpha is an index of reliability associated with the variation accounted for by the true score of the "underlying construct." Construct is the hypothetical variable that is being measured (Hatcher 1994).

Alpha coefficient ranges in value from 0 to 1 and may be used to describe the reliability of factors extracted from dichotomous (i.e., questions with two possible answers) and/or multi-point formatted questionnaires or scales (such as the one used here). The higher the score, the more reliable the generated scale is. Nunnaly (1978) has

indicated 0.7 to be an acceptable reliability coefficient but lower thresholds are sometimes used in the literature.

Tables 5.4, 5.5, and 5.6 show the results from the Cronbach's alpha calculated using the SAS program. Table 5.4 shows that the Cronbach's alpha coefficient for the four variables that measured developer satisfaction with their finished design was equal to .73. Table 5.5 shows that the Cronbach's alpha coefficient for the five variables that measured developer satisfaction with problem solving was equal to .84. Table 5.6 shows that the Cronbach's alpha for the five variables that measured developer satisfaction with their development process was equal to .81. These three numbers confirm the internal validity of the questionnaire.

Table 5.4 Cronbach's Alpha for the Four Items Measuring Developer Satisfaction with Their Finished Designs

Cronbach Coefficient Alpha

Variables	Alpha
Raw	0.722837
Standardized	0.738088

Cronbach Coefficient Alpha with Deleted Variable

	Raw Variables		Standardized Variables		
Variable	Deleted with Total	Correlation Alpha	with Total	Correlation Alpha	Label
SatDes1	0.558283	0.644086	0.594833	0.640782	SatDes1
SatDes2	0.568015	0.635821	0.609684	0.631890	SatDes2
SatDes3	0.521299	0.659294	0.503645	0.693604	SatDes3
SatDes4	0.441846	0.711273	0.419580	0.739660	SatDes4

Table 5.5 Cronbach's Alpha for the Five Items Measuring Developer Satisfaction with Their Problem Solving Process

Cronbach Coefficient Alpha

Variables	Alpha
Raw	0.838001
Standardized	0.839453

Cronbach Coefficient Alpha with Deleted Variable

	Raw Variables		Standardized Variables		
Variable	Deleted with Total	Correlation Alpha	with Total	Correlation Alpha	Label
PrbSlv1	0.714393	0.784705	0.710895	0.787876	PrbSlv1
PrbSlv2	0.579091	0.821894	0.574430	0.825432	PrbSlv2
PrbSlv3	0.502798	0.845642	0.516060	0.840756	PrbSlv3
Prbslv4	0.708887	0.785591	0.705752	0.789337	Prbslv4
PrbSlv5	0.717228	0.783050	0.714198	0.786937	PrbSlv5

Table 5.6 Cronbach's Alpha for the Five Items Measuring Developer Satisfaction with Their Development Process

Cronbach Coefficient Alpha

Variables	Alpha
Raw	0.805859
Standardized	0.817135

Cronbach Coefficient Alpha with Deleted Variable

	Raw Variables		Standardized Variables		
Variable	Deleted with Total	Correlation Alpha	with Total	Correlation Alpha	Label
Process1	0.422122	0.835773	0.424075	0.833415	Process1
Process2	0.686510	0.741979	0.696631	0.754274	Process2
Process3	0.715948	0.728796	0.728329	0.744360	Process3
Process4	0.528652	0.786976	0.524303	0.805557	Process4
Process5	0.671345	0.744133	0.683240	0.758417	Process5

5.2.2 Validation of the Judges

The judges' scores were validated using the paired two sample t-test and the pearson r calculation of the correlation coefficient between the two judges. The results are displayed in Table 5.7. The judges' average score and standard deviation are fairly consistent. The paired t-test shows a significant probability that the two judges' scores are related. Finally, the most definitive indicator is the correlation coefficient of .88 between the two judges scoring, which demonstrates that the judges' scores are highly correlated.

Table 5.7 Tests to Validate Judges Grading

	Judge1	Judge2	Validity Test
Average Score	82.14	84.57	
Standard Deviation	18.18	14.01	
Paired T-test Probability			0.09
Correlation Coefficient			0.88

5.2.3 Tests of Significance

Tables 5.8, 5.9, and 5.10 show the tests of significance that were performed on the overall data. As each hypothesis was a bi-variable statement, the t-test was selected as the appropriate test of significance to compare the means of the two variables analyzed.

Table 5.8 Non-CHIPS versus CHIPS – Means and t-test Probability

	Satisfaction with Final Design	Satisfaction with Problem Solving	Satisfaction with Development Process	Expert Judges Ranking
Non-CHIPS	6.19	5.91	5.84	78.82
CHIPS	6.07	5.76	5.81	86.58
Probability	N/A	N/A	N/A	0.012

Scores measured on 7-point Likert Scales (7=High) for columns 2-4
Scores measures on scale of 1-100 for column 5
The difference between the means is statistically significant at $p<.05$

Table 5.9 Non-CHIPS/No Change versus CHIPS/No Change – Means and t-test Probability

	Satisfaction with Final Design	Satisfaction with Problem Solving	Satisfaction with Development Process	Expert Judges Ranking
Non-CHIPS/No Change	5.96	5.71	5.64	79.41
CHIPS/No Change	6.21	5.95	6.04	85.09
Probability	0.049	0.042	0.003	0.077

Table 5.10 Non-CHIPS/Change versus CHIPS/Change – Means and t-test Probability

	Satisfaction with Final Design	Satisfaction with Problem Solving	Satisfaction with Development Process	Expert Judges Ranking
Non-CHIPS/Change	6.48	6.16	6.09	77.95
CHIPS/Change	5.96	5.60	5.62	87.94
Probability	N/A	N/A	N/A	0.048

The tests of significance show that the following hypotheses were supported (as summarized in Table 5.11):

- H1b - When system requirements remain constant throughout the development process, developers will be more satisfied with the finished system design when it is developed using CHIPS than when it is developed not using CHIPS.

- H2b - When system requirements remain constant throughout the development process, developers will be more satisfied with their problem solving capabilities developing a system using CHIPS than when they are not using CHIPS.

- H3b - When system requirements remain constant throughout the development process, developers will be more satisfied with the development process when using CHIPS than when not using CHIPS.

- H4a – Expert judges will rate the finished system design better when developers use CHIPS to develop the system than when developers do not use CHIPS.

- H4c - Expert judges will rate the finished system design better when system requirements change late in the development process and developers use CHIPS than when system requirements change late in the development process and developers do not use CHIPS.

Hypotheses H1a, H1c, H2a, H2c, H3a, H3c, and H4b were not supported.

Table 5.11 Summary of Hypotheses Supported and Not Supported

H1a	Developers will be more satisfied with the finished system design when it is developed using CHIPS than when it is developed not using CHIPS.	Not Supported
H1b	When system requirements remain constant throughout the development process, developers will be more satisfied with the finished system design when it is developed using CHIPS than when it is developed not using CHIPS.	**Supported**
H1c	When system requirements change late in the development process, developers will be more satisfied with the finished system design when it is developed using CHIPS than when it is developed not using CHIPS.	Not Supported
H2a	Developers will be more satisfied with their problem solving capabilities developing a system using CHIPS than when they are not using CHIPS.	Not Supported
H2b	When system requirements remain constant throughout the development process, developers will be more satisfied with their problem solving capabilities developing a system using CHIPS than when they are not using CHIPS.	**Supported**
H2c	When system requirements change late in the development process, developers will be more satisfied with their problem solving capabilities developing a system using CHIPS than when they are not using CHIPS.	Not Supported
H3a	Developers will be more satisfied with the development process when using CHIPS than when not using CHIPS.	Not Supported
H3b	When system requirements remain constant throughout the development process, developers will be more satisfied with the development process when using CHIPS than when not using CHIPS.	**Supported**
H3c	When system requirements change late in the development process, developers will be more satisfied with the development process when using CHIPS than when not using CHIPS.	Not Supported
H4a	Expert judges will rate the finished system design better when developers use CHIPS to develop the system than when developers do not use CHIPS.	**Supported**
H4b	Expert judges will rate the finished system design better when system requirements remain constant throughout the development process and developers use CHIPS than when system requirements remain constant throughout the development process and developers do not use CHIPS.	Not Supported
H4c	Expert judges will rate the finished system design better when system requirements change late in the development process and developers use CHIPS than when system requirements change late in the development process and developers do not use CHIPS.	**Supported**

5.3 Tests of Correlation

The previous section demonstrated the dependent variables that did or did not display a statistically significant result (as determined by the t-test) in order to show if the hypotheses were supported. In this section the strength of the correlation between the variables will be shown. There are several tests that can be used to determine the degree of association between variables. The correct test to use is chosen based on the type of variables involved (Rosnow and Rosenthal 2005).

For this research, the significant research questions were answered using a dichotomous independent variable (CHIPS versus non-CHIPS) and a series of continuous dependent variables (i.e., level of satisfaction and judges ranking). The most appropriate test of association, given those circumstances, is the point-biseral correlation, which is a special case of the product-moment r designed to handle such cases (Rosnow and Rosenthal 2005).

Tables 5.12, 5.13, and 5.14 show the result of the point biseral correlation calculation for the dependent variables versus the independent variable.

Table 5.12 Non-CHIPS versus CHIPS – Point Biseral R

Non-CHIPS vs. CHIPS	Satisfaction with Final Design	Satisfaction with Problem Solving	Satisfaction with Development Process	Expert Judges Ranking
Point Biseral R	-.05	-.06	-.01	.23

Table 5.13 Total No Late Stage Change, Non-CHIPS versus CHIPS – Point Biseral R

No Late Stage Change: non-CHIPS vs. CHIPS	Satisfaction with Final Design	Satisfaction with Problem Solving	Satisfaction with Development Process	Expert Judges Ranking
Point Biseral R	.097	.09	.14	.18

Table 5.14 Late Stage Change, Non-CHIPS versus CHIPS – Point Biseral R

Late Stage Change: non-CHIPS vs. CHIPS	Satisfaction with Final Design	Satisfaction with Problem Solving	Satisfaction with Development Process	Expert Judges Ranking
Point Biseral R	-.23	-.21	-.18	.26

5.4 Demographics

Figures 5.1, 5.2, 5.3, 5.4, and 5.5 display the demographic background of the subjects who participated in the experiment.

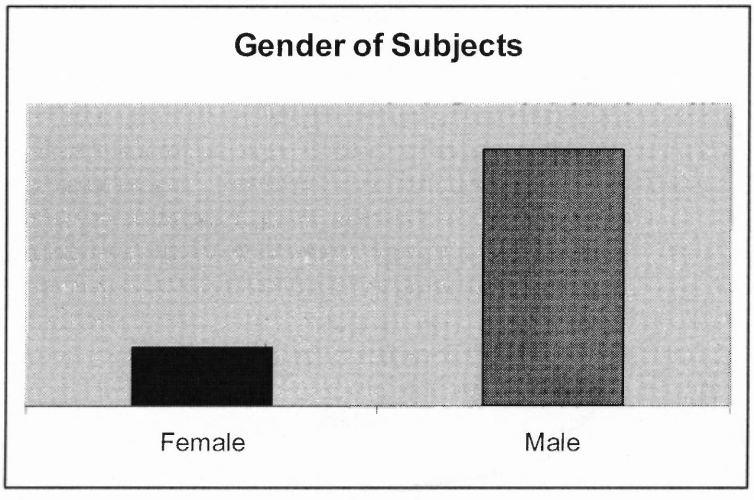


Figure 5.1 Gender of subjects.

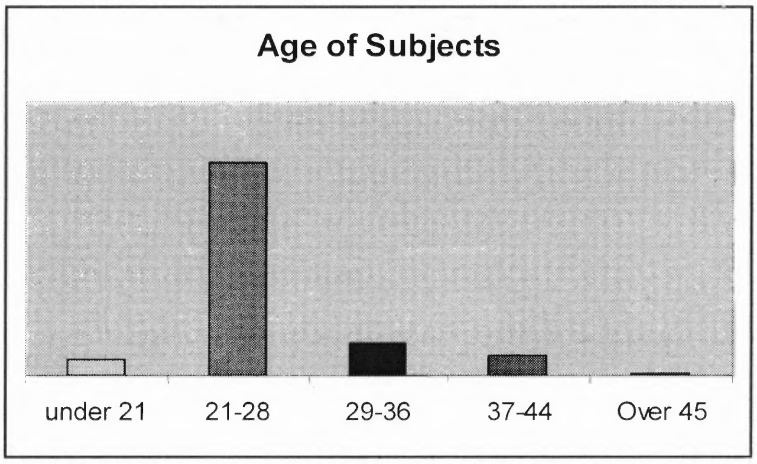


Figure 5.2 Average age of subjects.

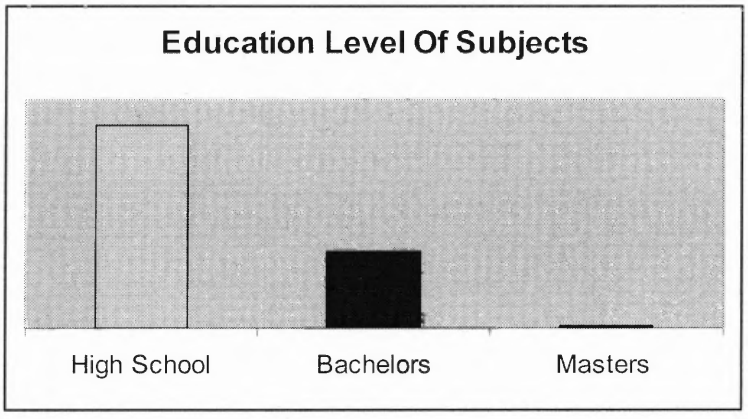


Figure 5.3 Average education level of subjects.

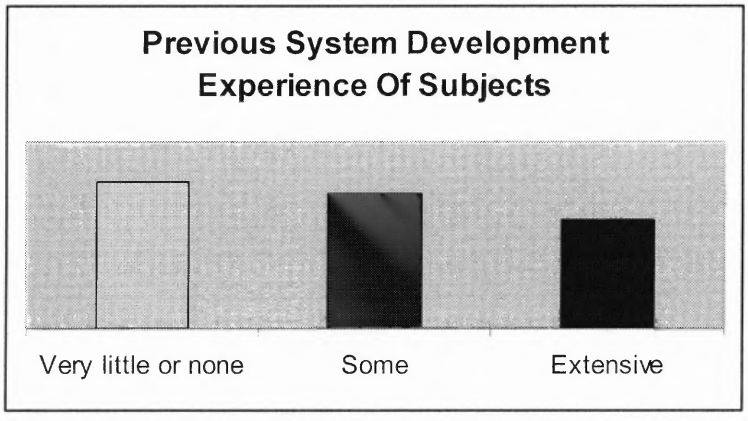


Figure 5.4 Previous system development experience of subjects.

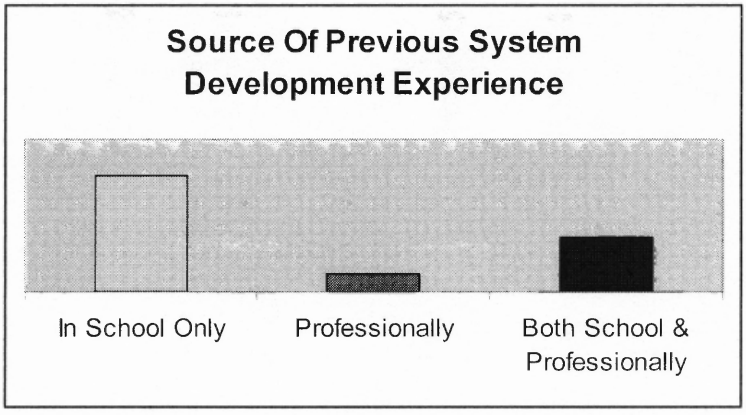


Figure 5.5 Source of previous system development experience.

CHAPTER 6
DISCUSSION AND CONCLUSION

6.1 Limitations of the Experiment

The discussion begins with some limitations of the experiment. The premise of the CHIPS model is that the system development process is dependent upon the contextual domain of the project and that the volatility and variance across and within domains cannot be normalized to a prescribed methodology or to a prescribed set of methodology fragments. A laboratory experiment, where an attempt is made to normalize and control the environment, appears counter-intuitive to that premise. Furthermore, a lab experiment could in no way simulate all of the circumstances that would be involved in a typical system development project.

Kaplan and Maxwell (1994) argue that the goal of understanding a phenomenon and its particular social and institutional context is largely lost when textual data are quantified. Given the premise of the model it appears that future data collection would be best accomplished in a qualitative manner (Myers 1997), perhaps through a case study approach or in a process whereby the results were triangulated with a quantitative method.

A second limitation of the experiment was the demographic background of the subjects, as displayed in Figures 5.1 – 5.5. The subjects were predominantly twenty-something college students whose previous development experience was based on knowledge gained through school work. A professional developer, whose background was based on real world experience, would surely bring a deeper understanding and

wider breadth of knowledge to the development process. The question arises as to how that background would affect the CHIPS process.

A third limitation is the nature of the experiment itself. The experiment task was given to the subjects as part of class assignment. Although the subjects had the option of completing an alternative assignment to the experiment, nearly all elected to participate in the experiment. Once the experiment process had begun, many students found the task to be laborious, especially when combined with the extra requirements of the CHIPS process. Since they had committed to the process and were thus required to complete it, it stands to reason that the subjects' attitudes towards the process would be affected. The manifestation of this limitation will become apparent as the results of the dependent variables are explored.

A final limitation of the experiment is that the subjects were asked only to perform design tasks. No implementation was performed. This brings into question whether the results obtained through this research would be successfully extrapolated across the other phases of the system development process.

6.2 Defense of the Experiment

Several arguments can be made as to the validity of and the reasoning behind the research approach taken. The primary goal of this research was to establish CHIPS as a legitimate system development model. A laboratory experiment provided a quick and efficient means by which to accomplish that goal. The results from the lab experiment did identify several areas where there were statistically significant differences between the use of the CHIPS model versus not using the CHIPS model. These results provide a metric against which future research can be measured.

Second, a lab experiment allowed for the initial research on the model to be conducted in a controlled environment. A controlled environment allowed the establishment of distinct cause and effect relationships through the manipulation of the independent variables (CHIPS vs. non-CHIPS and no changes to requirements vs. changes to requirements) and the array of dependent variables that were measured. The controlled environment also allowed the isolation of the independent variables as the key variables in the environment.

Third, a laboratory experiment provided a protocol that can be replicated. In the course of this research, the experiment was replicated four times with four different groups of subjects. The replication process allows for validation of the experiment in the present through several iterations and also in the future by future researchers.

Fourth, a laboratory experiment yields quantitative data. Quantitative data can be analyzed using inferential statistics. As demonstrated in the results chapter of this document, these tests allow the establishment of the likelihood that the cause and effect relationships were by chance or were statistically significant, and the measurement of the degree of the relationship between the independent and dependent variables.

Finally, one of the central goals of the CHIPS model is to establish it as a model that can easily be learned and used effectively by practitioners regardless of experience and education. By using inexperienced developers (i.e., students) as subjects, and by showing significant improvements through the utilization of the model by those in-experienced developers, it shows that the model can be learned and used by almost anyone.

6.3 Discussion of the Results

In the next few sections each of the dependent variables will be reviewed and analyzed. Conclusions will be drawn and inferences made as to the cause of the outcome of the various results.

6.3.1 Developers Satisfaction with the Finished System Design

As for the first dependent variable, developer satisfaction with the final design, there were no significant differences between CHIPS and non-CHIPS developers when their satisfaction was measured at the aggregate level. However, when the results were analyzed against the factor of whether late stage requirements changes were introduced, there were some significant differences. In particular, when late stage changes were not introduced, CHIPS developers were more satisfied with their design than those developers who did have late stage changes introduced.

At first, this result appears to be contrary to the intent of CHIPS. CHIPS is meant to make developers more satisfied with their finished products, particularly when there is uncertainty and change involved. The explanation for the results demonstrated here may be found in the deficiencies of the experiment discussed above. The experiment required that half the subjects be assigned a change in the requirements of the project late in the project development process. The changes came at a time when the subjects were close to finishing the project and were a total surprise to the subjects.

While a seasoned system developer would probably know that this is a common occurrence in a typical system development project, students would not. Students would look at the late stage change as an unmitigated burden and as an act of deception perpetrated upon them by their instructor. They had been told that they had to do x and

now they also had to do y. This argument is bolstered by Appendix I which displays some of the emails the researcher received from subjects who had received the late stage change. Many of the subjects were very angry at the fact that the requirements had changed (and many took the time to express that anger).

It is believed that the anger of the subjects who received the late stage change outweighed any increased satisfaction they may have gained by using the CHIPS model. This skewing of the results is reflected in all three dependent variables that measured developer satisfaction. Furthermore, it is believed that the impact of the decreased satisfaction of the late stage change subjects was so strong, that it affected the aggregate numbers that demonstrated overall satisfaction of CHIPS versus non-CHIPS developers.

Further weight can be made to this argument when the results of not introducing the late stage change are analyzed. With no late stage change, and thus no reflective anger, the CHIPS developers are clearly more satisfied with their finished designs than non-CHIPS developers.

6.3.2 Developers Satisfaction with their Problem Solving Process

The second dependent variable measured, developer satisfaction with their problem solving process, returned the same results as the developers satisfaction with their finished designs. Developers were not more satisfied with their problem solving process when this variable was measured overall showing CHIPS developers versus non-CHIPS developers and when late stage changes were introduced to the project. However, consistent with the previous variable, when no late stage changes were introduced, CHIPS developers were significantly more satisfied with their problem solving process.

Again, it is felt that these results are an indicator of the limitations of the experiment and that the anger of the subjects at the introduction of the late stage change outweighed any increased satisfaction that they may have felt by using the CHIPS model. When no late stage changes were introduced, a clearer indicator of the developers' increased satisfaction with the model is reported.

6.3.3 Developers Satisfaction with the Development Process

The third dependent variable, developer satisfaction with the development process, returned the same results as the previous two variables. Overall, CHIPS developers were not more satisfied with the development process than non-CHIPS developers and were significantly less satisfied when late stage changes were introduced to the project. However, consistent with the previous variables, when no late stage changes to the requirements were introduced, CHIPS developers were significantly more satisfied with the development process than non-CHIPS developers.

Again, it is felt that these results are an indicator of the limitations of the experiment and that the anger of the subjects at the introduction of the late stage change outweighed any increased satisfaction that they may have felt by using the CHIPS model. When no late stage changes were introduced, a clearer indicator of the developers' increased satisfaction with the model is reported.

6.3.4 Expert Judges Rating of the Finished Projects

The final dependent variable that was measured was the expert judges' rating of the finished designs. It is felt that the results obtained through the measurement of this variable provide the strongest and clearest indicator of the outcome of this research. This

is because the limitations of the experiment, which were outlined in section 6.1, are mitigated by the results obtained for this variable.

The expert judges were not effected by the factors that caused the results obtained from the other variables to be somewhat skewed. They had to do the same amount of work to grade each and every project, no matter what process the developers used. Furthermore, the expert judges were blinded as to what treatment the projects they were grading had been exposed to. The judges developed the grading criteria themselves and then used that grading criteria (appendix H) to grade the projects.

The judges were not allowed to confer with each other during the grading process. However, the validity of the results obtained is verified through the strong correlation between the grades assigned to the various projects by the two judges. (This correlation is shown in Table 5.7). It is because of these reasons that it is felt that the judges' ratings provided the most valid results of the experiment.

The results obtained from the measurement of the expert judges' ratings of the projects are clearly dissimilar to the results obtained from the other variables. The Expert judges rated the finished system designs significantly higher when developers used CHIPS to develop the system than when developers did not use CHIPS. The judges also found that when late stage changes were introduced to the developers, the developers created statistically significantly higher rated projects when they used CHIPS versus when they did not use CHIPS. This would make sense as the essence of CHIPS is to make the developers better able to adapt to change and thus provide a better product when there is change late in the development process. When there were no late stage changes for the developers, the CHIPS developers' projects were still ranked higher than

the non-CHIPS developers' projects, however the difference in the scores was not of a magnitude that would constitute a statistical significance.

Several questions arise from an analysis of this data. Deeper inspection shows that while scores were significantly higher for projects developed by CHIPS developers, the correlation coefficient between the scores and the use or non-use of CHIPS, while positive, is relatively weak, (as shown in Tables 5.12-5.14). While correlation is not necessarily an indicator of causation (Rosnow and Rosenthal 2005), the low correlation coefficient in this instance could be explained by the large standard of deviation found in each array of the judges scores (shown in Table 5.7). If the scores had been in a tighter range the Point Biseral R would have been a higher positive number.

Another question that arises from this data is whether these same results would be obtained from a live project conducted in a real development setting. Would developers working on a real world project, with its multitude of environmental factors, create better systems if they were to use the CHIPS model? That question cannot be answered from the data that was collected through this experiment. However, the results of this experiment show promise.

6.4 The Hevner, March, Park, and Ram Guidelines Revisited

As previously stated in Chapter 3, Hevner et al. (2004), list seven guidelines for performing design science research. The guidelines are not mandatory but should be addressed for a design science research project. Those guidelines are:

1. Creation of an innovative, purposeful artifact
2. A specified problem domain
3. Thorough evaluation of the artifact
4. The artifact must solve an unsolved problem in an innovative way
5. The artifact must be rigorously defined, formally presented, coherent, and internally consistent.
6. A problem space must be constructed and a mechanism is enacted to find a solution.
7. The results of the research must be communicated effectively.

Each of these guidelines has been addressed in Chapter 3. However, as part of the discussion and conclusion of this document, the guidelines will be revisited to confirm that they have been adhered to in the administration of this research. As was done in Chapter 3, the guidelines have been re-ordered here to follow a logical sequence of problem identification and solution.

Guideline # 1: A Specified Problem Domain Must Be Defined

A methodology in software engineering and IS development is defined as a recommended collection of phases, procedures, rules, techniques, tools, documentation, management, and training used to develop a system (Avison and Fitzgerald 2003, Cockburn 2002, Hoffer et al. 2005). There have been significant advances and changes

to methodological techniques over the last 30 years However, previous research has pointed out three distinct areas of concern:

1. No one methodology can claim to be the best method for every software project. (Cockburn 2002, Fitzgerald et al. 2003)
2. Present day developers are becoming increasingly discouraged with the traditional methods and their shortcomings (Avison and Fitzgerald 2003).
3. Those developers that have continued to use the traditional methods have modified and adapted those methods to meet the specificities of the project (Fitzgerald 1997, Fitzgerald et al. 2003).

Tailoring a methodology to the actual project development context has shown promise in addressing these concerns (Fitzgerald et al. 2003). There are three popular approaches to method tailoring. The contingency factors approach suggests that specific features of the development context should be used to select an appropriate methodology from a portfolio of methodologies. The "Method Engineering" approach (Brinkkemper 1996, Fitzgerald et al. 2003), constructs a methodology from a repository of "existing discrete predefined and pre-tested method fragments." (Fitzgerald et al. 2003). Using a method-engineering tool, software developers build a meta-method that is made up of fragments from popular development methodologies. The fragments are each designed to handle a particular contingency inherent to the software project. The final approach to method tailoring is the ad hoc approach used by practitioners to adapt methodologies to the context of their projects.

Both contingency factors and ME techniques have had little success in practical industry applications (Fitzgerald et al. 2003, Rossi et al. 2004). However, ad hoc methodology tailoring has been an implied concept for many years in industry (Fitzgerald 1997) and many popular methodologies such as the Rational Unified Process (Jacobson

et al. 1999) nearly mandate tailoring through modification and adaptation in order for them to work in a specific context.

Guideline #2: A problem space must be constructed and a mechanism is enacted to find a solution.

In accordance with this guideline, the Walls et al. (2004) framework was applied. Figure 3.1 shows a flowchart from Walls et al. demonstrating the relationships among the components of an IS design theory. The Walls model distinguishes between products, ("a plan of something to be done or produced") and processes, ("to so plan and proportion the parts of a machine or structure that all requirements will be satisfied" Walls et al. 2004). CHIPS is a model. Model artifacts are used to abstract and represent phenomena (Hevner et al. 2004) and would fall under the category of a product.

The Walls et al. process begins by choosing a kernel theory from the natural or social sciences. The kernel theory chosen for the model was General Systems Theory General systems theory is an interdisciplinary field that studies systems as a whole. It focuses on the complexity and the interdependence of the parts of a system (von Bertalanffy 1969).

One of the goals of general systems theory is to find common ground upon which scientific study can be conducted across several disciplines (von Bertalanffy 1969). Considering the hundreds of IS development methodologies (Fitzgerald et al. 2003) as "different disciplines" within the spectrum of IS development methodologies, it appears that general systems theory fits the goal of the CHIPS model quite well.

The next step in the Wall et al. process is to identify the meta-requirements of the artifact. Meta requirements describe the class of goals to which the (design) theory applies (Walls et al. 2004). The meta-requirements of the CHIPS model artifact are to

provide a means by which the best possible IS development methodology (or fragment thereof) is utilized at the best possible time, based on the circumstances, context, constraints of the project.

The next step in the Walls et al. formula is meta-design. Meta-design describes a class of artifacts hypothesized to meet the meta-requirements (Walls et al. 2004). For the previously defined meta-requirements, the meta-design was described as an artifact that provides the means by which to select the best possible IS development methodology (or fragment thereof) at the best possible time, based on the context, contingencies, and constraints of the project.

Guideline # 3: Creation of an innovative, purposeful artifact

In response to this guideline and guidelines 6.4.1 and 6.4.2 above, CHIPS, a collaborative, hierarchical, and incremental problem solving model was introduced. The model was borne of the concept that IS development is essentially a problem solving process (DeFranco-Tommarello and Deek 2002, Highsmith 2000), which allows the characterization of IS development methodologies as problem solving systems. These systems have several general system theory characteristics. They are open systems that interact with their outer (Simon 1996) environment, which means that they have the propensity for negative entropy. They also demonstrate the spectrum ranging from organized simplicity to disorganized complexity (Weinberg 1975). Finally, these systems all have a "system state" (Kuhn 1974), which represents the current condition of system variables (such as the current number of open, unsolved problems in the system).

General systems theory tells us that a system is complexes of elements standing in interaction (von Bertalanffy 1969). The CHIPS problem solving system model includes the following elements:

1. Problems - the difference between a goal state and the current state of the system (Hevner et al. 2004).
2. Problem Solving Processes - the tools, procedures, processes, etc. that are used to do the following (Deek et al. 1999):
    a. Define and understand problems.
    b. Plan solutions to problems.
    c. Implement solutions.
    d. Verify and present the results.
3. Solutions - The answer to or disposition of a problem (American Heritage Dictionary 2000).
4. People – These are the stakeholders (i.e., a person or organization that has a legitimate interest in a project or entity) of the development project.
5. Environment - the contingencies, constraints, rules, laws, etc. of the organization, people, technology, etc.

Guideline # 4: The artifact must be rigorously defined, formally presented, coherent, and internally consistent.

CHIPS is a model artifact. Model artifacts are used to abstract and represent phenomena (March and Smith 1995, Hevner et al. 2004). Like general systems theory, the intent of CHIPS is to provide a unification model across the disciplines of its domain. In the case of the CHIPS model, the "disciplines" refer to the plethora of system development methodologies and approaches. CHIPS is meant to model the elements of many system development projects and to be inclusive of many approaches and methodologies. Also like general systems theory, the CHIPS model seeks to unify the disciplines, not by

breaking the disciplines down, but by discovering the concepts that are isomorphic across the disciplines (von Bertalanffy 1969).

Constructs are the basic language of concepts from which phenomena can be characterized (March and Smith 1995). These concepts can then be combined into higher order constructions (i.e., models) used to describe tasks, situations, or artifacts (March and Smith 1995). The constructs of the CHIPS model include five classes of elements that are common across all system development projects. The five classes are: problems, solutions, people, problem solving mechanisms, and environmental concerns. Figure 3.3 and Figure 3.4 diagrams these concepts and their relationships

Using the constructs of the CHIPS model, the model was defined. The CHIPS model has the following phases, (as demonstrated in Figures 3.5 and 3.6), that are repeated in an iterative manner throughout the life of the project:

1. Describe:
    a. Define the current state of the project by:
        i. Analyzing the current environment.
        ii. Analyzing feedback from the previous iteration.
        iii. Identifying open problems (recall that problems are defined as problems that need to be solved and tasks that need to be completed).
    b. Dissect problems into sub-problems.
    c. Prioritize the open problems.
2. Problem Solve:
    a. Choose the highest priority problem.
    b. Apply a problem solving mechanism.

c. Choose a solution from the knowledge base of the project (fragments from other methodologies besides the base methodologies can also serve as solutions).

3. Prescribe:

a. Prescribe the next course of action for the project.

b. Record the problem as close/solved.

c. Update the knowledge base with the solution/action.

The describe phase is used to understand the current state of the project. It is a knowledge producing activity (March and Smith 1995). It includes analyzing the current environment and identifying circumstances that have changed since the last definition phase, analyzing feedback that was obtained from the previous iteration, analyzing and parsing the list of problems still open at the conclusion of the cycle, and adding to the list any new problems that can be identified. The list of open problems is then broken down into sub-problems, which are prioritized.

The problem solve phase is used to solve the highest priority problem in the list of open problems. If the problem is something simple, for instance a task that needs to be completed, then it can immediately pass to the next phase. However, if the problem is complex, then a problem–solving technique must be applied (i.e., brainstorming, Polya's method, etc) in order to collaboratively find a solution to the problem. The solution to the problem may be a methodology fragment. For instance, it may be determined that the best solution at this phase would be to build a prototype or to create an ER diagram.

The final phase is to prescribe. This is a knowledge using activity (March and Smith 1995). Using the knowledge gained during the previous two phases the next

course of action is prescribed. The problem that is solved by completing the action is now marked as a solved problem and the solution is recorded in the knowledge base.

CHIPS attempts to abstract methodologies to a common level. It assumes that all methodologies involve:

1. People working together collaboratively.
2. A set of unsolved problems and tasks.
3. An environment which defines the parameters and constraints of the project.
4. An algorithm for decomposing and prioritizing the unsolved problems and tasks.
5. A set of processes for solving the problems and completing the tasks in incremental steps.

By abstracting software development projects to the level and components suggested by CHIPS, the focus is placed on solving the problems and completing the tasks inherent to the project, rather than the employment of a methodology. This allows the developers to concentrate on improving and completing the core processes.

Methodology fragments can easily be selected, arranged, and inserted into the process. This allows the process to become as prescriptive or as adaptive as necessary and thus allow the project to meet the visibility requirements of management, yet adapt to unforeseen change.

The CHIPS model has several advantages:

1. Prescriptive/Evolutionary – CHIPS can be used to either prescribe a course of action or adapt to new issues that arise. It forces developers to identify problems (i.e., tasks), solutions, what people will participate, what problem solving mechanisms will be used for the project, and the current environment. IS development methodologies can be used to prescribe a course of action for the project and pre-defined problem solving mechanisms can be used to solve new problems as they arise.

2. Simple/Complex – it can be made as simple or complex as needed, dependent on the project.

3. Linear/Non-Linear – it can structure the tasks/problems in a linear fashion (such as in a typical "waterfall" approach) or in a non-linear fashion.

4. Dynamic and learning – the solutions class and the problem solving class are dynamic, expanding, learning, and remembering. In the future a CHIPS tool will have search features to facilitate matching solutions to tasks/problems.

5. It can encapsulate any development project and any set of development methodologies employed.

Chapter 3 demonstrates a methodology that could be employed through the use of the CHIPS model. It also takes a common methodology (i.e., the Rational Unified Process) and shows how that methodology could be improved by using the CHIPS model to blend RUP with the traditional waterfall approach to system development.

Guideline #5: The artifact must solve an unsolved problem in an innovative way

It is the goal of this research to present a more formalized model than the ad hoc methodology tailoring employed by practitioners. The inadequacies of the contingency factors approach are apparent (Fitzgerald et al. 2003). It is just not feasible or possible for all the developers in an organization to be familiar with all of the possible methodologies that would work best for a given situation (Fitzgerald et al. 2003). Plus as the contingent factors of the project change over time, so will the optimum methodology.

If method engineering is analyzed through the lens of general systems theory, it becomes apparent that it is both a reductionistic and mechanistic solution to the problem.

It is reductionistic in the sense that it attempts to solve the problem by reducing the phenomenon (the methodology) to its smallest component (method fragments) and analyzing the components. It is mechanistic because it attempts to build a whole meta-methodology from the sum of its parts, with no regard for the interrelationships of those parts. It was Aristotle who first stated, "The whole is more than the sum of the parts".

CHIPS is a holistic, anti-reductionistic, anti-mechanistic approach. It seeks to integrate (and thus fulfill the meta-requirements of the artifact) by identifying and capitalizing on the isomorphic characteristics of the IS development methodologies. In an open system negative entropy occurs through a process of decision making and feedback (von Bertalanffy 1969). Therefore it stands to reason that a model that focuses on problem solving would be one that leads to higher levels of order over time.

Guideline # 6: Thorough evaluation of the artifact

In accordance with this guideline the CHIPS model must have a thorough scientific evaluation. Chapter 4 of this document presents details of the research methods that were used to evaluate the model. In summary, a lab experiment was conducted to test the theory that CHIPS will enhance the system development process.

Guideline #7: The results of the research must be communicated effectively.

The purpose of this document is to effectively communicate the results of the research

6.5 Conclusions and Future Research

The results of this initial research, while promising, are significant for several reasons. First, mean scores were higher for CHIPS developers than they were for non-CHIPS developers for almost all of the dependent variables. This can tentatively be used as an indicator that CHIPS has a role as a legitimate model. Second, there were areas that showed a statistically significant improvement of CHIPS developers over non-CHIPS developers. Given the limitations of the experiment elaborated earlier, this could be an indicator that CHIPS has the potential to be a model that improves the development process. Finally, the results of this experiment give support to the argument that CHIPS can be used to integrate several system development methodologies, not by breaking those methodologies down into method fragments, but by identifying the isomorphic characteristics of those methodologies.

The separation of the IS development methodology community around heavy, proprietary tool oriented approaches versus "amethodological", light, open source approaches distracts us from more basic issues. None of the IS development methodologies that have been developed to date work well in the majority of situations. They all have to be refined and tailored extensively to the actual needs of the development context (Cockburn 2002, Fitzgerald et al. 2003). The existing accepted approaches to method tailoring (i.e., contingency and ME) have shortcomings as noted earlier.

The CHIPS model presented in this research directly addresses the problems inherent with other development methodology adaptation approaches. This general systems approach facilitates an IS community effort to normalize system development

methodologies. The adherence to design science guidelines lends itself to the legitimacy of the model. Practitioners who use this method will not have to learn methodologies that are not normalized. Thus, they will have a shorter learning curve to implement this technique versus the other method tailoring techniques. Our research community can work collaboratively to reduce ambiguity in methodologies by using the theoretical foundation presented here.

Future research is needed in several areas. First, additional lab experiments are needed to replicate this initial experiment, but with larger sample sizes. Second, field experiments are needed that will test the CHIPS model in a more realistic setting and against other popular methodologies and approaches. Finally, specific methodologies and instantiations of the CHIPS model need to be developed and evaluated accordingly.

APPENDIX A
CONSENT FORM

Appendix A is a copy of the consent form that was given to the experiment subjects.

CONSENT FORM

NEW JERSEY INSTITUTE OF TECHNOLOGY
323 MARTIN LUTHER KING BLVD.
NEWARK, NJ 07102

CONSENT TO PARTICIPATE IN A RESEARCH STUDY

TITLE OF STUDY: CHIPS

RESEARCH STUDY:

I, _____, have been asked to participate in a research study under the direction of Timothy Burns. Other professional persons who work with him as study staff may assist to act for him.

PURPOSE:

To test the satisfaction and software development effectiveness of using a collaborative, hierarchical, incremental, problem-solving model.

DURATION:

My participation in this study will last for 3 weeks.

PROCEDURES:

I have been told that, during the course of this study, the following will occur:

My team will be given a software problem to solve collaboratively. Following the completion of the tasks required, I am required to fill out a questionnaire and participate in a debriefing session.

My grade will be based on my ability to follow directions, the quality of my performance on the specific tasks, and my participation level.

PARTICIPANTS:

I will be one of about 160 participants to participate in this trial.

EXCLUSIONS:

I will inform the researcher if any of the following apply to me: N/A

RISK/DISCOMFORTS:

I have been told that the study described above may involve the following risks and/or discomforts:

There are no known risks or discomforts.

There also may be risks and discomforts that are not yet known – N/A.

CONFIDENTIALITY:

Every effort will be made to maintain the confidentiality of my study records. Officials of NJIT will be allowed to inspect sections of my research records related to this study. If the findings from the study are published, I will not be identified by name. My identity will remain confidential unless disclosure is required by law.

PAYMENT FOR PARTICIPATION:

I have been told that I will receive $0 compensation for my participation in this study.

CONSENT AND RELEASE:

I fully recognize that there are risks that I might be exposed to by volunteering in this study which are inherent in participating in any study; I understand that I am not covered by NJIT's insurance policy for any injury or loss I might sustain in the course of participating in the study.

I agree to assume and take on myself all risks and responsibilities in any way associated with this activity. I release NJIT, its trustees, agents, employees and students from any and all liability, claims and actions that may arise as a result of my participation in the study. I understand that this means that I am giving up my right to sue NJIT, its trustees, agents and employees for injuries, damages or losses I may incur.

RIGHT TO REFUSE OR WITHDRAW:

I understand that my participation is voluntary and I may refuse to participate, or may discontinue my participation at any time with no adverse consequence. I also understand that the investigator has the right to withdraw me from the study at any time.

INDIVIDUAL TO CONTACT:

If I have any questions about my treatment or research procedures that I discuss them with the principle investigator. If I have any addition questions about my rights as a research subject, I may contact:Robin-Ann Klotsky, Executive Director of Research and Development at (973) 596-5227.

SIGNATURE OF PARTICIPANT

I have read this entire form, or it has been read to me, and I understand it completely. All of my questions regarding this form or this study have been answered to my complete satisfaction. I agree to participate in this research study.

Subject: Name: _____

Signature:_____

Date: _____

SIGNATURE OF READER/TRANSLATOR IF THE PARTICIPANT DOES NOT READ ENGLISH WELL

The person who has signed above,
_____, does not read English well,
I read English well and am fluent in (name of the language)
_____, a language the subject understands well.
I have translated for the subject the entire content of this form. To the best of my knowledge, the participant understands the content of this form and has had an opportunity to ask questions regarding the consent form and the study, and these questions have been answered to the complete satisfaction of the participant (his/her parent/legal guardian).

Reader/Translator Name: _____

Signature: _____

Date: _____

SIGNATURE OF INVESTIGATOR OR RESPONSIBLE INDIVIDUAL

To the best of my knowledge, the participant,
_____, has
understood the entire content of the above consent form, and comprehends the study. The participants and those of his/her parent/legal guardian have been accurately answered to his/her/their complete satisfaction.

Investigator's Name: _____

Signature: _____

Date: _____

APPENDIX B
PRE-EXPERIMENT QUESTIONNAIRE

Appendix B is a copy of a questionnaire that was given to the subjects prior to the experiment. The purpose of the questionnaire was to capture subjects' demographic information.

PRE-EXPERIMENT DEMOGRAPHIC QUESTIONNAIRE

Please Complete the Following Questionnaire:

1. Name:
2. Gender: M____ F____
3. Age: <=21____ 21-28____ 29-36____ 37-44____ >44____
4. Occupation: _____
5. Years at Job: _____
6. College Degrees Earned:

   Bachelors: _____ Discipline:_____

   Masters: _____ Discipline:_____

   PhD: _____ Discipline:_____

7. Previous Software Development Experience:

   Very little or none: _____
   Some: _____
   Extensive: _____

8. Previous Software Development Experience:

   In School Only: _____
   Professionally: _____
   Both: _____
   Neither: _____

APPENDIX C
EXPERT JUDGES TASK LIST

Appendix C is a copy of the list of tasks that was given to the expert judges to perform in the experiment.

EXPERT JUDGES TASK LIST

You will perform the following roles and complete the following tasks for the CHIPS experiment:

1) Act as "Users"
   a) Create the general requirements for an ecommerce web site
      i) Subjects will create a detailed design only – no implementation
      ii) Must be user oriented not system oriented
      iii) Subjects must be able to complete the design within 2 weeks
   b) Must present requirements to users
   c) Have a significant change to the requirements that can be introduced to the users late in the development process
   d) Act as a point of contact during the experiment

2) Determine how the subjects will present their finished product to you
   a) Perhaps can use Word to build a mock up

3) Act as "Expert Judges"
   a) Develop a tool to evaluate and rank the finished designs
      i) Heuristic Evaluation?
   b) Evaluate and rank the finished designs

4) Make a presentation to the subjects to initiate the experiment:
   a) You will remain outside the presentation room while the Experiment Coordinator (EC, i.e., Tim) introduces the experiment to the subjects and takes care of the initial administrative tasks.
   b) You will then be introduced to the subjects. You are to play the role of a marketing firm that sells e-commerce web sites.
   c) You will then have 15-20 minutes to explain the requirements of the web site and the grading criteria to the subjects. You also must allocate time to answer any questions.
      i) The subjects will have a copy of the document "Subject Task List" which includes grading criteria and deliverables/due dates.
      ii) Please review the requirements and the grading criteria in anticipation of subjects' questions.
      iii) DURING THE PRESENTATION PLEASE DO NOT ANSWER ANY QUESTIONS RELATED TO THE ADMINISTRATION OF THE EXPERIMENT. ANSWER REQUIREMENTS QUESTIONS ONLY.
      iv) The EC will NOT answer any questions pertaining to the web site requirements. He will only answer questions related to the administration of the experiment.

d) Once the 20 minute period is up, you will exit the presentation room. If there are still any unanswered questions, the subjects will be asked to submit the question to the experiment email address.

5) During the 2 week period of the experiment:
   a) You will periodically receive emails from the EC asking for clarification or additional information on the requirements. Please confer with each other and respond back to the EC with your answers.
   b) Under no circumstances are you to communicate directly with the subjects.

6) Grading Period:
   a) Each judge will be given a group of projects to grade INDIVDUALLY based on the grading criteria. Please grade and submit to the EC.
   b) Please do not discuss or confer with each other on project grading. YOU MAY ONLY CONFER WITH THE EC AT THIS POINT.
   c) Each judge should complete a separate grading form for each project.
   d) Once you have finished grading the projects you will submit the grade sheets to the EC.
   e) You will then be given a second set of projects to grade. Repeat steps b-d above. Please grade and submit to the EC.

APPENDIX D

SUBJECT TASK LIST

Appendix D is a list of tasks that was given to the subjects to perform in the experiment.

SUBJECT TASK LIST

1. You are a participant in an experiment being conducted on software development methods.

2. You will be asked to act in the role of a software development team and to develop a software product design.

3. You have been assigned to a group. You will choose one member of your group to be the leader. The leader is to handle the administrative duties of the group and is responsible for collecting and submitting pertinent information for the group. The group leader will receive additional points in the final grading of the project.

4. Do not discuss the task that you are to complete with anyone else besides the members of your group.

5. Direct any questions to NJITXP@optonline.net.

6. Any questions that are asked will be shared with all experiment participants.

7. Your final grade for the project will be determined by the grade you receive for the quality of the software product design and your participation level in the experiment. Team members will grade each other's participation.

eCOMMERCE PROJECT REQUIREMENTS

PRIMARY CONTACT: **NJITXP@optonline.net**
DUE DATE: *November 23, 2005*

Purpose: Simulate web site software that could be sold to and utilized by various eCommerce retail companies.
Opportunity: Increase the visibility and productivity of an eCommerce retail company which could lead to revenue growth and a greater market share.
Execution: Develop a web site template which is consumer friendly and simulates back end functions to manage the customers' purchases.

Use WORD and/or Excel to:

1) **Introduction:** Write at least a one page, organized introduction of an e-commerce software product. This product should be capable of placing orders and tracking relevant information for a retail company. This introduction should be written as if you want to sell the product to the reader.

2) **Hierarchy Chart:** Develop and draw a hierarchy chart to represent the organization of an e-commerce web site used to sell retail products.

3) **User Interface:** Create 6 - 10 "mock" user interface screens that allow the user to simulate the purchase and shipping of retail products.
    a. The screens should mirror the look of a web site.
    b. Include a narrative of how each screen will function.
    c. Use a fresh and contemporary color scheme and graphics.
    d. Usability:
        i. Consistent font and image lay out
        ii. Novice user language - "friendly"
        iii. Sort and search functions
        iv. Drop down data entry options
        v. Data entry error messages and other relevant inventory control messages
    e. Functionality: Users can…
        i. Place and ship orders
        ii. Log onto the web site to pull up their personal profile
        iii. Access other relevant web links

4) **Back-End Reports:** Create templates of various reports that could be used by the retail company to evaluate these suggested areas:
    a. Customer list
    b. Orders
    c. Returns
    d. Inventory levels and controls
    e. Sales

f. Web trends

5) **Conclusion:** Write at least a one page conclusion outlining the benefits of your program. This conclusion should be written as a "sales pitch".

Grading Criteria

Introduction: (5 points)
Organization:
Consistency of terminology:
Grammar/semantics:
Level of interest:

Hierarchy chart: (15 points)
Usability:
Organization:
Friendliness:
Visual appeal:
Functionality:
Originality:
Creativity:
Ability to traverse easily through site
Relevant information:
Lack of extraneous information:

Data entry screens: (40 points)
Usability:
Organization:
Visual appeal:
Friendly language:
Relevant field names and field sizes:
Sort and search functions:
Drop down options:
Functionality:
Originality:
Buttons and links:
Creativity:
Sort and search
Ability to traverse easily through site
Links to other relevant web sites:
Collection of required data:

Reports: (25 points)
Usability:
Readability:
Friendliness:
Use of titles and headers:
Functionality:
Distribution of information amongst various reports:

Organization of data fields:
Includes relevant data fields:

Conclusion: (5 points)
Consistency of language and terms:
Use of dynamic, convincing language:
Grammar/semantics:

Adherence to Project Requirements: (10 points)

Deliverables & Due Dates

All deliverables should be emailed to: NJITXP@optonline.net

11/03 - Select a team leader and a team name.

The team leader should then send an email to NJITXP@optonline.net :

1. Place the team number and name in the subject line (e.g. "Team 1 – THUNDERBIRDS")
2. CC all members of the team
3. Include in the body of the email:
    a. Team number
    b. Team name
    c. Team leader's name and email address
    d. Each team member's name and email address

11/09 – Hierarchy chart

11/16 – User entry screen mock-ups

11/23 – Completed Projects

11/30 – Submit the following: (these forms will be emailed to each person individually on 11/24)

Peer grading form
Post-Experiment Questionnaire
Debriefing Forms

APPENDIX E
PROBLEM SOLVING DOCUMENTATION

Appendix E is a copy of the documentation that was given to subjects on various problem solving techniques.

PROBLEM SOLVING DOCUMENTATION GIVEN TO ALL SUBJECTS

POLYA'S FOUR-STEP PROCESS

1. UNDERSTAND THE PROBLEM

- Can you state the problem in your own words?
- What are you trying to find or do?
- What are the unknowns?
- What information do you obtain from the problem?
- What information, if any, is missing or not needed?

2. DEVISE A PLAN

The following list of strategies, although not exhaustive, is very useful.

- Look for a pattern.
- Examine related problems, and determine if the same technique can be applied.
- Examine a simpler or special case of the problem to gain insight into the solution of the original problem.
- Make a table.
- Make a diagram.
- Write an equation.
- Use guess and check.
- Work backward.
- Identify a subgoal.

3. CARRY OUT THE PLAN

- Implement the strategy or strategies in step 2, and perform any necessary actions or computations.
- Check each step of the plan as you proceed. This may be intuitive checking or a formal proof of each step.
- Keep an accurate record of your work.

4. LOOK BACK

- Check the results in the original problem. (In some cases this will require a proof.)
- Interpret the solution in terms of the original problem. Does your answer make sense? Is it reasonable?
- Determine whether there is another method of finding the solution.
- If possible, determine other related or more general problems for which the techniques will work.

Brainstorming

Brainstorming is "a conference technique by which a group attempts to find a solution for a specific problem by amassing all the ideas spontaneously by its members" (Alex Osborn)

Brainstorming is a process designed to obtain the maximum number of ideas relating to a specific area of interest.

Brainstorming is a technique that maximizes the ability to generate new ideas.

Brainstorming is where a group of people put social inhibitions and rules aside with the aim of generating new ideas and solutions.

Brainstorming is a time dedicated to generating a large number of ideas regardless of their initial worth.

Brainstorming is a part of **problem solving** which involves the creation of new ideas by suspending judgment.

Brainstorming is the creation of an optimal state of mind for generating new ideas.

Brainstorming is the free association of different ideas to form new ideas and concepts.

Brainstorming process

1. Define and agree on the objective.
2. Brainstorm ideas and suggestions having agreed upon a time limit.
3. Categorize/condense/combine/refine.
4. Assess/analyze effects or results.
5. Prioritize options/rank list as appropriate.
6. Agree action and timescale.
7. Control and monitor follow-up.

SWOT Analysis - Strengths, Weaknesses, Opportunities and Threats

SWOT is a frequently used management tool, useful for reflection, decision-making and appraising options. It is particularly useful because of its simplicity, the way in which it takes seconds to set up, and can be easily explained to others and therefore used as a group exercise.

How to use it

The central idea is to take whatever you're wishing to consider and to look at it in terms of 4 areas:

Strengths - what does this idea have as advantages? What does it bring to the organization (or yourself?) What other things are linked to it that would be advantageous?

Weaknesses - What are the intrinsic problems with the idea? What are the associated costs (financial, resources, management time etc)

Opportunities - What avenues could this open up? How does this idea fit with the existing strategy, or could it bring new ideas into the ongoing strategic development? The opportunities part should be those things outside the actual issue, and will often be outside the organization itself

Threats - What are the dangers of adopting this approach? How will others see the change?

Brainstorm around these themes for a few minutes - it doesn't matter really if some things go in different or multiple categories, the important thing is to get the ideas out there and work through them.

Once you've got your Strengths, Weaknesses, Opportunities and Threats worked out, you can begin to consider if the Strengths and Opportunities outweigh the Weaknesses and Threats. You may see that there is an immediate threat, which means that the idea is not viable, but try to think a little deeper to see if the idea can be changed in some way to minimize this threat.

APPENDIX F
CHIPS DOCUMENTATION

Appendix F is a copy of the documentation that was given to subjects explaining the CHIPS model.

CHIPS DOCUMENTATION GIVEN TO SUBJECTS

CHIPS – Collaborative Hierarchical Incremental Problem Solving

CHIPS is a model that can be used in software development. Using the CHIPS model, software development teams work collaboratively to develop software. Each software project is treated as a set of problems that must be solved in order to successfully complete the project. Starting at a very high level, problems are identified, dissected into smaller problems, and arranged hierarchically. The problems are then solved in a bottom-up fashion using small incremental steps. CHIPS also requires that the development team pre-negotiate problem solving techniques that can be used to solve unexpected problems.

The CHIPS process can be stated as follows:

1. As a group, select two or three problem solving techniques and become familiar with how those techniques work. Use the techniques to solve unexpected problems that come up during the development process and to help with the CHIPS process.

2. As a group, identify the primary problem areas for the project. (Please note that the word problem can be defined as a problem or a task).

3. Dissect the problem areas into smaller tasks/problems and arrange the tasks hierarchically. Repeat this process for several iterations until problems have been broken out to a reasonable level.

4. Assign responsibility for the smaller tasks/problems to team members

5. As a task/problem is completed, record that it has been completed, who completed it, what (if any) problem solving technique was used to solve the problem (or complete the task), and what the solution was (if a problem solving technique was used).

6. Repeat this process at regular intervals throughout the development life cycle.

Project: Wash the Car

Team: Jim, Fred, John, Robin

Problem Solving Tools

P1 Polya Method
P2 Brainstorming
P3 SWOT Analysis

Tasks/Problems In Hierarchical Order

Sts	Seq#	Task	Assigned To	PS Tool	Solution
C	1.0	Get Tools	All		
C	1.1	Hose	Robin		
C	1.2	Soap	Fred		
C	1.3	Sponge	John		
C	1.4	Towel	Jim		
C	2.0	Rinse Car	Robin		
	3.0	Apply Soap			
C	3.1	Roof	Jim		
	3.2	Hood	Jim		
C	3.3	Front	John		
	3.3.1	Tar on driver's door		P1,P2,P3	Use tar remover
	3.4	Sides	Fred		
	3.5	Back	John		
C	4.0	Rinse Car	Robin		

APPENDIX G
LATE STAGE CHANGE DOCUMENT

Appendix G is a copy of the documentation that was given to subjects informing that a late stage change was to be made to the requirements of the project.

LATE STAGE CHANGE DOCUMENTATION

Due to a recent change in our business we need to make a change to the original requirements that were given to you. We apologize for making a change this late in the process but it is crucial to our business. You will receive extra compensation (i.e., extra points) for your efforts in implementing the change.

You were originally instructed to create a mock e-commerce web site for the product of your choice. We now would like the web site to be able to handle multiple products and industries. We would like you to add a configuration tool that can be used by the system administrator to change the look and processing capabilities of the web site depending on the product. You should include mock up screen(s) and a narrative description of the tool. You should also update your hierarchy chart to reflect how the tool is integrated with the rest of the web site.

Please include this additional information with your final submission due 11/23/05.

Thank you,

Jaclyn and Liz

APPENDIX H
POST-EXPERIMENT QUESTIONNAIRE

Appendix H is the final questionnaire given to the subjects after they completed their project.

POST-EXPERIMENT QUESTIONNAIRE

Final Questionnaire

Please answer all of the following questions and then click the submit button at the bottom

Name: _____

Last 4 Digits of SS#: _____

Email Address: _____

Experiment Team#: _____

Date: _____

1. I am very satisfied with the quality of my group's solution.

Strongly Agree — **Undecided** — **Strongly Disagree**

○ 7 ○ 6 ○ 5 ○ 4 ○ 3 ○ 2 ○ 1

2. I am *NOT* confident in the group's final solution.

Strongly Agree — **Undecided** — **Strongly Disagree**

○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7

3. I am very committed to my group's final solution.

Strongly Agree — **Undecided** — **Strongly Disagree**

○ 7 ○ 6 ○ 5 ○ 4 ○ 3 ○ 2 ○ 1

4. My group's problem solving process was efficient.

Strongly Agree — **Undecided** — **Strongly Disagree**

○ 7 ○ 6 ○ 5 ○ 4 ○ 3 ○ 2 ○ 1

5. My group's problem solving process was coordinated.

Strongly Agree — Undecided — Strongly Disagree

○ 7 ○ 6 ○ 5 ○ 4 ○ 3 ○ 2 ○ 1

6. My group's problem solving process was unfair.

Strongly Agree — Undecided — Strongly Disagree

○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7

7. My group's problem solving process was confusing.

Strongly Agree — Undecided — Strongly Disagree

○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7

8. My group's problem solving process was satisfying.

Strongly Agree — Undecided — Strongly Disagree

○ 7 ○ 6 ○ 5 ○ 4 ○ 3 ○ 2 ○ 1

9. My group's process to develop the web site was efficient.

Strongly Agree — Undecided — Strongly Disagree

○ 7 ○ 6 ○ 5 ○ 4 ○ 3 ○ 2 ○ 1

10. My group's process to develop the web site was coordinated.

Strongly Agree — Undecided — Strongly Disagree

○ 7 ○ 6 ○ 5 ○ 4 ○ 3 ○ 2 ○ 1

11. My group's process to develop the web site was unfair.

Strongly Agree — Undecided — Strongly Disagree

○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7

12. The final solution and formal reports do not reflect my inputs.

Strongly Agree			Undecided			Strongly Disagree
1	2	3	4	5	6	7

13. My group's process to develop the web site was confusing.

Strongly Agree			Undecided			Strongly Disagree
1	2	3	4	5	6	7

14. My group's process to develop the web site was satisfying.

Strongly Agree			Undecided			Strongly Disagree
7	6	5	4	3	2	1

APPENDIX I

JUDGES GRADING CRITERIA

Appendix I is a copy of the criteria that the expert judges used to grade the projects.

JUDGES GRADING CRITERIA

Introduction: (4 points)
- Organized
- Convincing
- Proper use of grammar/semantics
- Interesting and creative

Hierarchy Chart: (12 points)
- Properly organized
- Complete
- Attractive layout
- Creative

Customer Interface Screens: (40 points)

Design – How it looks:
- Organized, consistent layout
- Attractive layout – Appropriate use of graphics
- Customer friendly language – appropriate field sizes, etc…
- Creative/Original

Content – How it works:
- Organized
- Creative & Original
- Customer friendly
- Performs all necessary functions - Collects required data
- Eliminates extraneous information
- Narratives well written
- Ability to traverse easily through site- appropriate use of buttons, links, sorts, & searches

Back End System: (30 points)

"Mock" company screen
- User friendly
- Functional
- Well written narrative

Reports:

How they look:
- Visually appealing
- Readable
- Friendly
- Use of titles and headers
- Creative/Original

How they work:
- Clarity of information
- Information properly distributed amongst various reports
- Organized format of data
- Includes relevant data

Excludes extraneous data

Conclusion: (4 points**)**

Organized
Convincing
Proper use of grammar/semantics
Interesting and creative

Adherence to Project Requirements: (10 points)

Internal Grading Guidelines

Major Point Deductions (4 point reductions):

Data Entry Screens:

- Homepage
  - No logo/company name
  - No links to other sections of the website
- Product sales
  - No list of potential products
  - No pictures of products
- Purchasing and shipping
  - Did not ask for customer address
  - Did not ask for credit cards

Reports:

- Missing any of the five suggested reports
- Completely lacking graphics

Adherence to project requirements:

- Missing any major portion of requirement
- Missing change item – **6 point deduction**

Minor Deductions (2 point reductions):

Data Entry Screens:

- No drop down entries offered
- Homepage
  - No option for customer registration
  - No option for returning customer log in
  - No "Contact Us" or link to company homepage
- Product sales
  - No detailed product screen (information about the specific product chosen)
- Purchasing
  - Not asking for separate shipping address
  - No confirmation number for the order offered
- Narratives – missing narratives (2 points per screen)

Reports:

- Missing important data field(s) – **One point each**
- Including extraneous data field(s) – **One point each**

APPENDIX J

SAMPLES OF IRATE E-MAILS SENT BY SUBJECTS

Appendix J shows excerpts from emails sent by subjects to the experiment coordinator. The emails highlight the subjects displeasure at the introduction of a change to the project requirements being made late in the development process.

SAMPLES OF IRATE E-MAILS SENT BY SUBJECTS

The following paragraphs are excerpts from emails sent by subjects to the experiment coordinator. The subjects were upset that they had been assigned a change to the project requirements late in the project cycle.

Letter #1:

YOU HAVE GOT TO BE KIDDING ME ! ! ! ! !

By my displeasure, you can be assured that I received the notice of the project scope change. I must inform you that I will be out of town for the next few days and **will not** be able to discuss this change with my partner until Sunday evening at the earliest. That does not give us much time to develop a plan, design a mock-up screen, and write a narrative.

John

Letter # 2:

Excuse me? "Please make sure that all of your team members have received this email."?? I believe that's YOUR RESPONSIBILITY, not mine, not their's either. After all, WE all agreed to participate in YOUR lousy experiment. And now it's OUR responsibility to make sure that WE get the link so we can answer YOUR questionaire? Go pound sand, buddy! Your experiment sucks.

...Kelly

Letter #3:

To Experiment UNcoordinator:
You can stop now... I understand it's part of your 'experiment' to be intentionally misleading and confusing. First you have our report, then you don't. At this point, I honestly don't care. And I'm certainly not worried about my grade. You've wasted too much of my time already so I won't bother with your questionnaires. The results are probably skewed already.

...KS

REFERENCES

Abrahamsson, P., Warsta, J., Siponen, M., Ronkainen, J. (2003) "New Directions on Agile Methods: A Comparative Analysis", IEEE.

Adler, PS, (1995) "Interdepartmental Interdependence and Coordination: The Case of the. Design/Manufacturing Interface", Organization Science, 6, 2, 147-167.

Agarwal, R., Prasad, J., Tanniru, M., Lynch, J. (2000) "Risks of Rapid Application Development", Communications of the ACM.

Agarwal, R., Lucas, H. (2005) "The Information Systems Identity Crisis: Focusing on High-Visibility and High Impact Research", MIS Quarterly 29, 3, 381-398, September 2005.

Ahl, V., Allen, T. F. H. (1996) *Hierarchy Theory, a Vision, Vocabulary and Epistemology*, Columbia University Press.

Alavi, M.(1984) "An assessment of the prototyping approach to information systems development", Communications of the ACM, 27, 6.

Allen, J. J. and O. Hauptman (1987) "The Influence of Communication Technologies on Organizational Structure," Communication Research, 5, 14, 575-587.

The American Heritage Dictionary of the English Language (2000), Fourth Edition, Houghton Mifflin Company.

Ankolekar, A., Herbsleb, J., Sycara, K. (2003) "Addressing Challenges to Open Source Collaboration with the Semantic Web", 3rd Workshop on Open Source, Software Engineering ICSE'03 International Conference on Software Engineering Portland, Oregon May 3-11, 2003

Arasu, A., Cho, J., Garcia-Molina, H., Paepcke, A., Raghavan, S. (2001) "Searching the Web" ACM Transactions on Internet Technology, 1, 1, August 2001, Pages 2–43.

Ashby, W.R. (1947) "Principles of the Self-Organizing Dynamic System", Journal of General Psychology, 37,125-128.

Ashby, W.R. (1956) *Introduction to Cybernetics*, Chapman & Hall.

Augustin, L., Bressler, D., Smith, G. (2002) "Accelerating Software Development through Collaboration" ICSE '02, May 19-25, 2002, Orlando, Florida, USA. ACM 2002.

Avison, D., Fitzgerald, G. (2002) *Information Systems Development: Methodologies, Techniques and Tools*, Third Edition McGraw-Hill/Irwin.

Avison, D., Fitzgerald, G. (2003) "Where Now for Development Methodologies?", Communications of the ACM, 46,1 2, 79-82.

Ajzen,I. (1985) "From Intention to Actions: a Theory of Planned Behavior, in: J. Kuhl, J. Beckmann (Eds.), Action Control: From Cognition to Behavior", Springer-Verlag, New York, NY, 11-39.

I.

Baskerville, R., Levine, L., Pries-Heje, J., Ramesh, B., & Slaughter, S. (2002). Balancing Quality and Agility in Internet Speed Software Development. Paper presented at the International Conference On Information Systems (ICIS), Barcelona.

Bayer, S., Highsmith, J. (1994) "RADical Software Development", American Programmer Magazine, June 1994.

Beck et al. http://www.agilemanifesto.org/

Beck, K. (2000) *Extreme Programming Explained*, Pearson Education.

Becker, B. , Rasala,R. ,Bergin,J. , Shannon,C. , Wallingford, E. (2001) ACM SIGCSE Bulletin , Proceedings of the thirty second SIGCSE technical symposium on Computer Science Education February 2001, 33, 1.

Benbasat, I., and Zmud, R. (2003) "The Identity Crisis Within the IS Discipline: Defining and Communicating the Discipline's Core Properties," *MIS Quarterly* (27:2), June 2003, 183-194.

Boehm, B. (1988) "A Spiral Model of Software Development and Enhancement", IEEE, May 1988,61-72.

Boehm, B. (1999) "Escaping The Software Tar Pit: Model Clashes and How to Avoid Them", ACM SIGSOFT, 24,1, January 1999.

Boehm, B., Huang, L. (2003) "Value Based Software Engineering : Reinventing 'Earned Value' Monitoring and Control", ACM SIGSOFT, 28, 2, March 2003.

Booch, G. (1994) *Object-Oriented Analysis and Design with Applications*, 2ndEd, Benjamin/Cummings, Redwood City, Calif.

Booch, G. (2001) "Developing The Future", Communications of the ACM, 44, 2, March 2001, 119-121.

Botkin, John C. (1994) "Customer Involved Participation as Part of the Application Development Process AM/FM International.

Boulding, K. E. (1956) "General Systems Theory - The Skeleton of Science," Management Science, 2(3), 197-208.

Brinkkemper, S. (1996) *Method Engineering: Engineering of Information Systems Development Methods and Tools*, Elsevier Science B.V.

Brooks Jr., F. (1987) "No Silver Bullet, Essence and Accidents of Software Engineering", Computer Magazine.

Burns, T., Klashner, R. (2005) "A Cross-Collegiate Analysis of Software Development Course Content", Proceedings of the 6th Conference on Information Technology Education, Newark, NJ, USA, 333-337.

Choi, S., Deek, F., (2002) "Extreme Programming Too Extreme?", New Jersey Institute of Technology.

Chandrasekaran, B. (1990) "Design Problem Solving: A Task Analysis," AI Magazine, 4, 59-71.

Cheatham, T., Crenshaw, J. (1999) " Object-oriented vs. Waterfall Software Development", Proceedings of the 19th annual conference on Computer Science, p.595-599, April 1999, San Antonio, Texas, United States.

Cronbach, L. J. (1951). Coefficient Alpha and the Internal Structure of Tests. Psychometrika. 16, 297-334.

Chu-Carroll, M., Sprenkle, S. (2000) "Coven: Brewing Better Collaboration through Software Configuration Management", ACM SIGSOFT.

Coad, P. Yourdon, E. (1991) *Objected Oriented Analysis*, Yourdon Press Computing Series, NJ.

Coad, P., Lefebvre, E., De Luca, J. (1999) *Java Modeling In Color with UML: Enterprise Components and Process*, Prentice Hall.

Cockburn, A. (1999) "Characterizing People as Non-Linear, First-Order Components in Software Development", http://www.CrystalMethodologies.org.

Cockburn, A. (2002) *Agile Software Development*, Addison-Wesley.

Daft, R. L. and Lengel, R. H. (1986) "Organizational information requirements, media richness and structural design", Management Science, 32, 5 (May. 1986), 554-571.

Deek, F., McHugh, J. (2003) *Computer-Supported Collaboration with Applications to Software Development*, Klower Academic Publishers, Boston.

Deek, F.P., Turoff, M., McHugh, J.(1999) "A Common Model for Problem Solving and Program Development", Journal of the IEEE Transactions on Education, 42, 4, 331-336.

DeFranco-Tommarello, J., Deek, F.P (2002) "Collaborative Software Development: A discussion of Problem Solving Models and Groupware Technologies" Proceedings of the 35th Annual Hawaii International Conference on System Sciences. IEEE.

DeSanctis, G. (2003) "The Social Life of Information Systems Research: A Response to Benbasat and Zmud's Call for Returning to the IT Artifact," *Journal of the AIS* (4:7), December 2003, 360-376.

DeSousa, C., Redmiles, D., Dourish, R. (2003) "Breaking the Code, Moving Between Private and Public Work in Collaborative Software Development" Group '03. ACM.

Domino, M., Collins, R., Hevner, A., Cohen, C. (2003) "Conflict in Collaborative Software Development", SIGMIS Conference 03, April 10-12, 2003, Philadelphia, Pennsylvania Copyright 2003 ACM.

Feller, J., Fitzgerald, B. (2000) "A Framework Analysis of the Open Source Software Development Paradigm", The 21st International Conference in Information Systems (ICIS 2000), pp. 58- 69.

Forte, G., McCulley, K., eds. (1991) "CASE Outlook: Guide to Products and Services" CASE Consulting Group, Lake Oswego, Ore.

Fishbein, M., Ajzen,I. (1975) *Belief, Attitude, Intention, and Behavior: An Introduction to Theory and Research*, Addison-Wesley, Reading, MA.

Fitzgerald, B. (1997) "The use of systems development methodologies in practice: A field study", The Information Systems J. 7, 3, 201–212.

Fitzgerald, B., Russo, N., O'Kane, T. (2003) "Software Method Tailoring at Motorola", Communications of the ACM, 46,4.64-70.

Fowler, M. (2002) "The New Methodology", http://martinfowler.com/articles/newMethodology.html, Accessed May 18, 2005, 8pm.

Ginat, D. (2002) "On Varying Perspectives of Problem Decomposition", SIGCSE '02, February 27- March 3, 2002, Covington, KY, ACM.

Greefhorst, D. (2000) "Feature Driven Software Logistics" Software Engineering Research Centre.

Hardgrave, B. (1997) "Adopting Object-Oriented Technology: Evolution or Revolution", The Journal of Systems and Software, 37, 19-25.

Harrison, W., Osher, H., Tarr, P. (2000) "Software Engineering Tools and Environments: A Roadmap", Proceedings of the conference on the future of Software engineering. Limerick, Ireland.

Hatcher, L. (1994) A step-by-step approach to using the SAS(R) system for factor analysis and structural equation modeling. Cary, NC: SAS Institute.

Hevner, A., March, ST, Park, J., and Ram, S. (2004) "Design Science Research in Information Systems,". MIS Quarterly, 28, 1, 75-105.

Highsmith, J. (1997) "Messy, Exciting, and Anxiety Ridden: Adaptive Software Development". American Programmer, X, 1; January 1997.

Highsmith, J. (2000) *Adaptive Software Development - A Collaborative Approach to Managing Complex Systems*, Dorset House Publishing, New York, NY.

Highsmith, J. (2002) "What Is Agile Software Development?", CROSSTALK The Journal of Defense Software Engineering, October 2002.

Hirsch, M. (2002) "Making RUP Agile", OOPSLA 2002 Practitioner Report. ACM 2002.

Hoffer,J., George, J., Valacich, J. (1999) *Modern Systems Analysis & Design*, Second Edition, Addison-Wesley.

Hoffer,J., George, J., Valacich, J. (2005) *Modern Systems Analysis & Design*, Fourth Edition, Addison-Wesley.

Howard, A. (2002) "Rapid Application Development: Rough and Dirty or Value-for-Money Engineering?", Communications of the ACM, 45, 10.

Ives, B., Parks, M. S., Porra, J., and Silva, L. (2004) "Phylogeny and Power in the IS Domain: A Response to Benbasat and Zmud's Call for Returning to the IT Artifact," *Journal of the AIS* (5:3), March 2004, 108-124.

Jacobson, I., Booch, G., Rumbaugh, J. (1999) *The Unified Software Development Process*, Addison-Wesley.

Johnson, R.A. (2000) "The Ups and Downs of Object-Oriented Systems Development", Communications of the ACM, 43, 10.

Johnson, R.A., Hardgrave, B., Doke, E. (1999) "An Industry Analysis of Developer Beliefs About Object-Oriented Systems Development", The Data Base for Advances in Information Systems. Winter 1999, 30, 1.

Johnson-Eilola (2002) "Open Source Basics: Definitions, Models, and Questions", SIGDOC'02, October 20-23, 2002, Toronto, Ontario, Canada, Copyright 2002.

Kaplan, B. and Maxwell, J.A. (1994) "Qualitative Research Methods for Evaluating Computer Information Systems," in *Evaluating Health Care Information Systems: Methods and Applications,* J.G. Anderson, C.E. Aydin and S.J. Jay (eds.), Sage, Thousand Oaks, CA, 45-68.

Kuhn, A. (1974) T*he Logic of Social Systems*, San Francisco: Jossey-Bass.

Kruchten, P. (2001) Book Review http://www.therationaledge.com

Kruchten, P. (2000) *The Rational Unified Process: An Introduction*, 2nd ed. Addison-Wesley.

Leffingwell, D. (2001) "Features, Use Cases, Requirements, Oh My!", The Rational Edge.

Littlewood, B., Strigini, L. (2000) "Software Reliability and Dependability: A Roadmap", ACM.

Maner, W. (1997) http://csweb.cs.bgsu.edu/maner/domains/RAD.htm.

March, S. and Smith, G, (1995) "Design and Natural Science Research on Information Technology." Decision Support Systems 15 (1995): 251 - 266.

Martin, J. (1991) *Rapid Application Development*, New York: Macmillan Publishing Company.

Mockus, A., Felding,R., Herbsleb, J. (2002) "Two Case Studies of Open Source Software Development: Apache and Mozilla." ACM Transactions on Software Engineering and Methodology, 11, 3, July 2002, 309–346.

Mumford, E. (1981) "Participative Systems Design: A Structure Method." Systems, Objectives, Solutions 1,1, 5-19.

Myers, Michael D. (1997) "Qualitative Research in Information Systems," *MIS Quarterly,* Vol. 21, No. 2, pp. 241-242. *MISQ Discovery*, archival version, June 1997, http://www.misq.org/misqd961/isworld/. *MISQ Discovery*, updated version, http://www.auckland.ac.nz/msis/isworld/.

Naumann, J. D., Jenkins, A.M. (1982) "Prototyping: The New Paradigm for Systems Development." MIS Quarterly 6, 3, 29-44.

Naur, P., Randell,B. (eds.). (1969) "Software Engineering: A Report on a Conference sponsored by the NATO Science Committee." NATO, http://www.cs.ncl.ac.uk/old/people/brian.randell/home.formal/NATO/nato1968.PDF

Newell, A., Simon, H. (1972) *Human Problem Solving*, Prentice Hall.

Nunnaly, J. (1978) *Psychometric Theory*, New York: McGraw-Hill.

Nuseibeh, B., Kramer, J., Finkelstein, A., (2003) "Viewpoints: Meaningful Relationships Are Difficult!", IEEE.

Orlikowski, W.J. & Barley, S.R. (2001). Technology and Institutions: What Can Research on Information Technology and Research on Organizations Learn from Each Other? MIS Quarterly, 25, 145-165.

Palmer, S., Felsing, J. (2002) *A Practical Guide to Feature-Driven Development*, Prentice Hall.

Perens, B. (1999) *Open Sources: Voices from the Open Source Revolution,* 1st Edition. Chapter 1, O'Reilly.

Pirolli, P. (1999) Cognitive engineering models and cognitive architectures in human-computer interaction, Chapter 15 in *Handbook of Applied Cognition*, Durso, F.T., (ed.), John Wiley & Sons, NY, 443-477.

Probasco, L. (2000) "Ten Essentials of RUP."" The Rational Edge, December 2000 http://www.therationaledge.com/content/dec_00/f_rup.html

Polya,G. (1957) *How to Solve It*, 2nd ed., Princeton University Press, ISBN 0-691-08097-6.

Racoon, L.B.S. (1997) "Fifty years of progress in software engineering", ACM SIGSOFT Software Engineering Notes, 22, 1, January 1997.

Raymond, E. (2003) "The Cathedral and the Bazaar", 1998, www.tuxedo.org accessed July 20, 2003.

Robey, D. (2003) "Identity, Legitimacy and the Dominant Research Paradigm: An Alternative Prescription for the IS Discipline: A Response to Benbasat and Zmud's Call for Returning to the IT Artifact," *Journal of the AIS* , 4, 7, December 2003, 352-359.

Rogers, E. (1995) *The Diffusion of Innovations*, Fourth ed., Free Press, New York, NY.

Rosnow, R. L., & Rosenthal, R. (2005). *Beginning behavioral research: A Conceptual Primer* (5th ed.). Upper Saddle River, NJ: Prentice Hall.

Rossi, M., Ramesh, B., Lyytinen, K., and Tolvanen, J. (2004) "Managing Evolutionary Method Engineering by Method Rationale," Journal of the AIS, 5, 9.

Royce, W.W. (1970) "Managing the Development of Large Software Systems: Concepts and Techniques", Proceedings of IEEE, Westcon.

Saff, D., Ernst, M. (2004) "An Experimental Evaluation of Continuous Testing During Development", *ISSTA '04,* July 11–14, 2004, Boston, Massachusetts, USA Copyright 2004 ACM.

Schwaber, K., http://controlchaos.com.

Schwaber, K., Beedle, M. (2002) *Agile Software Development with SCRUM*, Prentice Hall. New Jersey.

Shannon, C.E., Weaver, W. (1949) *The Mathematical Theory of Communication,* University of Illinois Press.

Simon, Herbert, (1962) "The Architecture of Complexity" in *Proceedings of the American Philosophical Society*, 106, 1962, 467-482.

Simon, H. (1996) *The Sciences of the Artificial*, Third Edition. Cambridge, MA, MIT Press.

Sommerville, I. (1996) *Software Engineering*, Addision-Wesley.

Steiner, I.D. (1972) *Group Process and Productivity,* Academic Press: New York.

Sutherland, J., Heuval, W. (2002) "Enterprise Application Integration Encounters Complex Adaptive Systems: A Business Object Perspective", HICSS 2002.

Swift, M. (1989) "Prototyping in IS Design and Development", Journal of Systems Management. July 1989. 14-20.

Truex, D., Baskerville, R., Travis. J. (2000) "Amethodical Systems Development: The Deferred Meaning of Systems Development Methods", Journal of Accounting, Management, and Information Technologies, 10, 53-79.

Truex, D., Avison, D. (2003) "Method Engineering: Reflections on the Past and Ways Forward", Ninth Americas Conference on Information Systems.

Umpleby, Stuart A. (2001) "Two Kinds of General Theories in Systems Science", Online Proceedings of the American Society for Cybernetics 2001 Conference, Vancouver, May 2001. http://www.asc-cybernetics.org/2001/Umpleby.htm

von Bertalanffy, L. (1928) *Kritische theorie der Formbildung*, Borntraeger.

von Bertalanffy, L. (1969) *General System Theory*, Braziler, New York.

Walls, J. G., Widmeyer, G. R., and El Sawy, O. A. (1992) "Building an Information System Design Theory for Vigilant EIS",. Information Systems Research, 3, 1, March 1992, 36-59.

Walls, J. G., Widmeyer, G. R., and El Sawy, O. A. (2004) Assessing Information System Design Theory in Perspective: How Useful was our 1992 Initial Rendition?", Journal of Information Technology and Application (JITA), 6, 2, 43-58.

Wartsa, J., Abrahamsson, P. (2003) "Is Open Source Software Development Essentially an Agile Method?" ICSE'03 International Conference on Software Engineering Portland, Oregon May 3-11, 2003.

Weinberg, G. (1975) *An Introduction to General Systems Thinking* (1975 ed., Wiley-Interscience)

Wheeler, D. (2003) "Why Open Source Software / Free Software (OSS/FS)? Look at the Numbers!", http://www.dwheeler.com.

Whiting, R. (1998) "Development in Disarray", Software Magazine, September, 20.

Wood, J., Silver, D. (1995) *Joint Application Development*, 2nd ed., New York: Wiley.

WWW, http://www.agilemanifesto.org.

WWW, http://cgi.omg.org/news/pr97/umlprimer.html.

WWW, http://crystalmethodologies.org.

WWW, http://www.computer.org/Seweb/Dynabook/HumphreyCom.htm 2002

WWW, http://www.dsdm.org.

WWW, Cutter. (2000, October). Light Methodologies Best for E-business Projects. Cutter Consortium. http://cutter.com/consortium/research/2000/crb001003.html

WWW, www.necsi.org.

WWW, Rational Unified Process:Best Practices for Software Development Teams http://www.rational.com.

WWW, http://www.SOFTWAREmag.com/archive/2001feb/CollaborativeMgt.html, Accessed January 15, 2006, 3pm.

WWW, WWW, http://www.standishgroup.com/sample_research/chaos_1994_1.php, Accessed July 23, 2004, 11am .

WWW,http://www.step-10.com.

WWW, http://www.surgeworks.com (DSDM).

WWW, http://www.xprogramming.com.

Yourdon, E., Constantine, L.L (1979) *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 1st edition, Prentice-Hall.