A Comparative Study of Iterative Prototyping vs. Waterfall Process Applied To Small and Medium Sized Software Projects

by

Eduardo Málaga Chocano

B.S., System Engineering (1996)
National University of Engineering, Lima, Peru

SUBMITTED TO THE SYSTEM DESIGN AND MANAGEMENT PROGRAM IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF IN ENGINEERING AND MANAGEMENT AT THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2004

© 2004 Massachusetts Institute of Technology. All rights reserved.

Signature of Author . . . . . .
Eduardo Malaga Chocano
System Design & Management Program

Certified by . . . . . .
Olivier de Weck
Thesis Supervisor
Robert N. Noyce Career Development Professor
Assistant Professor of Aeronautics & Astronautics and Engineering Systems

Accepted by . . . . . .
Tomas J. Allen
Co-Director, LFM/SDM
Howard W. Johnson Professor of Management

Accepted by . . . . . .
David Simchi-Levi
Co-Director, LFM/SDM
Professor of Engineering Systems

A Comparative Study of Iterative Prototyping vs. Waterfall Process Applied To Small and Medium Sized Software Projects

by

Eduardo Malaga Chocano

B.S., System Engineering
National University of Engineering of Peru, 1996

SUBMITTED TO THE SYSTEM DESIGN AND MANAGEMENT PROGRAM ON APRIL 22, 2004 IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCES IN ENGINEERING AND MANAGEMENT

Abstract

After Royce introduced the Waterfall model in 1970, several approaches looking to provide the software development process with a formal framework have been elaborated and tested. While some of these followed the sequential line of thought presented by Royce and Boehm, other methodologies have suggested the use of iterations since early stages of the lifecycle as a mean to introduce feedback and gain understanding.

This thesis takes a look at both types of approaches in an attempt to identify their strengths and weaknesses and based on this build criteria to recommend a particular approach or approach's elements for a given a set of conditions.

Literary research and interviews with experienced project managers were conducted to identify software development issues and understand how these can be better addressed by the use of development methodology. Based upon this research a system dynamics model was developed. This model was used to simulate the effects that different approaches might have on a software project under similar and different situations.

Analysis of the data suggests that, under certain conditions, iterative approaches are more effective to increase productivity due to learning and therefore more likely to finish earlier. They also promote a better distribution of time diminishing developers' idle time. On the other hand, sensitivity analysis shows that sequential approaches are more stable in terms of duration and quality and therefore a less risky option when initial conditions are uncertain.

Thesis Supervisor: **Olivier de Weck**
Robert N. Noyce Career Development Professor
Assistant Professor of Aeronautics & Astronautics and Engineering Systems

Acknowledgements

This work wouldn't have been possible without the help of many people. First of all, I want to thank my thesis advisor, Professor Olivier de Weck, who gently accepted to supervise my work. His guidance and support was fundamental to the development of this study.

Thank to all the colleagues in Peru that kindly shared with me their experiences developing software projects. Your help was fundamental to interpret previous studies and build the model that was developed as part of this thesis.

I also want to thank the faculty, staff and classmates in the SDM program. You helped to make of this a great experience that I will never forget. A special thank to Christos Sermpetis and Keen Sing Lee, my teammates in System Project Management coursework, whose work contributed greatly to the study presented in Appendix E.

Finally, I'd like thank my family and all my friends who, despite the distance, helped me and gave me all the support I needed to write this thesis and successfully finish my studies at MIT.

Gracias Totales!

Table of Contents

List of Figures

List of Tables

Chapter I: Introduction

I.1. Motivation

> "A quarter of a century later software engineering remains a term of aspiration. The vast majority of computer code is still handcrafted from raw programming languages by artisans using techniques they neither measure nor are able to repeat consistently."[1]

Since its creation in the late 1950s, software systems have dramatically evolved in terms of size, complexity, presence and importance. As a result of this evolution, different issues related to the development of software have emerged. One of the most common critiques is the appreciation about how unpredictable software projects are[2].

Software engineering, emerged as a discipline in 1968 at the NATO Software Engineering Conference, has been studying mechanisms to address the challenges that the increasing size and complexity of software has brought[3]. Efforts have covered a wide range of categories including improvements in programming languages, development techniques, development tools and development methodologies.

The waterfall model, one of the first software development methodologies developed in the 1970s, is one of the most remarkable examples of engineering applied to software. One of the most important contributions of this model was the creation of a culture of "thinking"

[1] Gibbs W. Wayt, "Software's Chronic Crisis", Scientific American, September 1994: p. 86.
[2] Although success in software can have different definitions, in terms of conformity with initial budget and delivery time, studies conducted in by the Software Engineering Institute (SEI) at Carnegie Mellon University, suggest that schedule and cost targets are usually overrun in organizations without formal management methodologies. In "Software project management for small to medium sized projects", Prentice-Hall, 1990, Rakos says that initial estimations in software projects are 50% to 100% inaccurate.
[3] Kruger Charles, "Software Reuse", ACM Computing Surveys, Vol. 24, No. 2, June 1992: p 132.

before “coding”. In the 1980’s, and in the absence of other approaches, this model became a development standard. This model, with some variations, is still widely used in the software industry today.

In opposition to this approach, some other models embracing iterative development cycles and the use of prototyping emerged in the 80’s and 90’s. In 2001, some of the most recognized leaders of these methodologies decided to share their common thoughts and propose a new way to develop software. Their ideas are published in what they called the Agile Manifesto[4].

Which approach is better and under which conditions is one approach more appropriate? These are questions that haven’t been unanimously answered. In the meantime, more methodologies are being created without a careful study of what we can learn from past experience.

I.2. Objectives and Hypothesis

The first objective of this thesis is to identify the main strengths and weaknesses of iterative cycle based models vs. sequential-based models applied to small and medium sized software projects. A second objective is to measure the impact these features have in the management of projects. A third objective is to understand under which conditions each of these approaches is recommended. Finally, a fourth objective is to study the new trends in this field and propose recommendations regarding their use.

In order to accomplish these objective this thesis poses several hypothesis. The central hypothesis is that software development methodologies have a significant impact in success of software project. A second hypothesis is that most software development methodologies can be categorized in two main groups: sequential phase models and iterative cycle models. A third hypothesis is that some features of these methodologies can be isolated in order to

[4] “Manifesto for Agile Software Development” (http://agilemanifesto.org/)

study their impact on developing projects. Finally, a fourth hypothesis is that some of these features can be combined in order to propose new approaches that can improve the management of software projects.

Although many aspects are equally applicable to all kinds of software development, flexibility in goals' definition and prioritization are significantly different regarding the context where systems will be used. In this regard, this thesis focuses only on the development of business application software within business organizations.

I.3. Approach

The research approach to be used for this study includes as a first step a literary review. This review focuses on three aspects: the story and evolution of the development methodologies, critical analysis and studies of these methodologies, and the use of system dynamics as a tool to simulate behavior in software projects. A second approach is to conduct interviews with project manager experts in order to understand how these methodologies are used in practice. Using the information from the literary research and the interviews, models to simulate different scenarios and understand the behavior of the two approaches will be made. An analysis from the results obtained from the simulations will provide the basis to evaluate the hypothesis and accomplish the goals of this thesis.

Chapter 1 provides an introductory review about this study including goals and hypothesis. Chapter 2 provides an overview of some key software concepts including a brief description of the most important software development methodologies. Chapter 3 provides and explanation of the methodology research used in this study and developed in the next chapter. Chapter 4 includes a phase decomposition of sequential and iterative methodologies, an overview of the interviews conducted to project managers, and the explanation of the system dynamics model developed to compare sequential and iterative methodologies. Chapter 5 show the data obtained running the model described in the previous chapter. Finally, Chapter 6 and 7 provides a summary and the conclusions of this study (see Fig. 1).

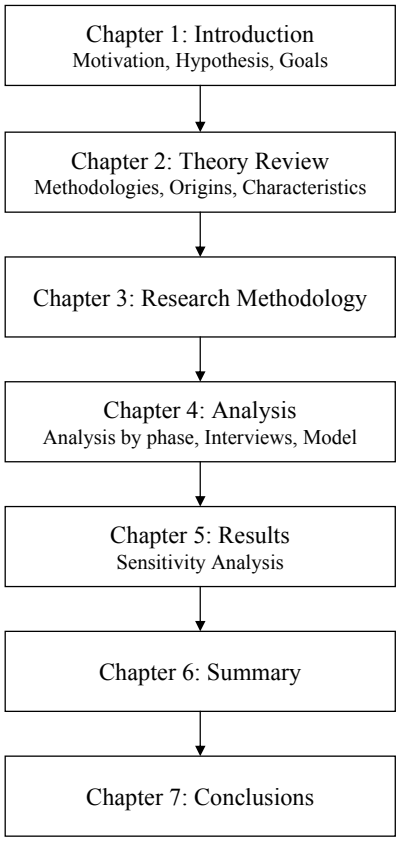

Figure 1. Thesis Roadmap

Chapter II: Theory Review

This chapter discusses some underlying software concepts as well as the evolution and definitions of some of the most often used development methodologies. This review will provide the theoretical basis necessary to understand the analysis illustrated in the following chapters.

II.1. Software Concepts

II.1.1. Abstraction

Abstraction, defined as a succinct description that suppresses unimportant details and emphasizes relevant information, is one of the most unique characteristics of software[5]. It is abstraction what allows us to develop systems out of ideas and concepts.

Abstraction gives us the ability to manage concepts that don't have any specific instance or physical representation. This ability plays a key role in software because, in contrast to any other engineering field, software products are just sets of information structures stored in some physical media that are able to perform a pre-defined set of functions under some specific conditions. In the absence of physical representations, traditional techniques used by other engineering fields to design, model, build, test and maintain software systems cannot be directly applied to software.

Abstract sophistication has allowed software to evolve and produce larger and more powerful systems. Layers of abstraction have been increasingly added to software to

facilitate design by eliminating the complexity of handling physical devices. High-level programming languages are the result of this evolution. Likewise, software technologies such as structured programming and objected oriented are also built over abstract concepts[6].

II.1.2. Flexibility

People attribute to software a very controversial feature: flexibility. In fact, experience shows that small systems can be easily modified by the people who developed them. This seemingly ease to introduce changes has been reinforced by the wide range of functionality that programming languages offer these days. However, although many changes can be quickly introduced, integration analysis and testing should always be conducted along with the changes[7].

Flexibility has shown not always to be a good feature. If not properly managed, it can lead to significant delays in the lifecycle of a project.

II.1.3. Learning Curve

According to Brooks, if a task can be partitioned among many workers with no communications among them, then men are interchangeable [8]. However, software's characteristics, such as abstraction and flexibility, require a high level of communication among the development team members and, hence, men are rarely interchangeable.

---

[5] Kruger Charles, "Software Reuse", ACM Computing Surveys, Vol. 24, No. 2, June 1992: pp. 134-136.
[6] For more information about the importance of abstraction in software see Kruger Charles, "Software Reuse", ACM Computing Surveys, Vol. 24, No. 2, June 1992
[7] Studies of failures in complex system show how chains of apparent unrelated insignificant events could trigger terrible consequences. See papers by Leveson Nancy, "Medical Devices: The Therac-25" , University of Washington, 1995, and "Systemic Factors in Software-Related Spacecraft accidents", Massachusetts Institute of Technology, 2001 (available from http://sunnyday.mit.edu)
[8] Brooks Frederick, "The Mythical Man-Month", Addison Wesley Longman, 1995: p. 16.

More insight about Brook's Law was provided by Madnick and Tarek's work (see Fig. 2). Their model showed that adding new people to a late project generates a decrease in productivity, restraining the new employees from reaching a high or even normal level of productivity within the remaining time of the project[9].

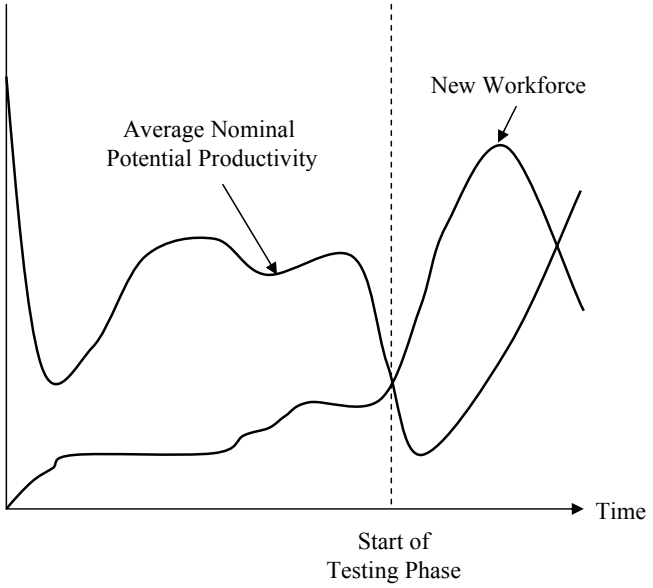

Figure 2. Aggressive manpower acquisition, Source: Adapted from "Software Project Dynamics" (Madnick and Tarek, 1982)

Moreover, according to his studies, ratios between the best and the worst programmer can be about 10:1 in terms of productivity and 5:1 in terms of speed. Although, as is discusses in the next section, metrics for software haven't shown to give consistent results to improve the development, still these numbers are an indication of how important the experience factor is for software development.

These two aspects highlight the benefits of an integrated, experienced team to achieve success in software development. Once again, traditional approaches to increase productivity,

[9] A study of Brook's Law and the conditions for its validity is discussed in Madnick Stuart, Abdel-Hamid Tarek K., "The dynamics of software project scheduling: a system dynamic perspective", Center for Information Systems Research, Alfred P. Sloan School of Management, 1982: pp. 111-122.

such as incorporating new people into a project to shorten its duration, are not applicable for software.

II.1.4. Metrics

Having no physical representations, metrics to quantify aspects of software are also hard to define. Although some attributes such as performance, memory allocation, etc., can certainly be quantified some other important aspects such as size, complexity, friendliness and overall quality tend to be highly influenced by subjective factors.

One of the most common measures is Lines of Code (LOC). Conte, Dunsmore, and Shen define a Line of Code as "any line of program text that is not a comment or blank line, regardless of the number of statements or Fragments of statements on the line"[10]. Although this is a very simple and objective metric, it usually fails to provide accurate insight regarding the size of a program. Nevertheless, factors such as programming language, modularity, reuse, complexity and even the programmer style can seriously impact the line of codes of a program.

Another metric for size is the Function Count, which represents a module or logical unit. A function is an abstraction of part of the tasks that the program is to perform. Again, unless strict rules about how a code can be split in modules and functions are made, different persons will likely interpret this metric in different ways.

Size metrics for data structures are less subject to interpretation. Number of entities, attributes and relationships provide a good insight about the size of a data structure. Likewise, size of raw data, size of indexes, and size of space used in a database, are also good metrics to determine the characteristics of a database.

[10] Conte S. D., Dunsmore H. E., and Shen V.Y., "Software Engineering Metrics and Models", Benjamin/Cummings, 1986: p. 35.

Aspects such as productivity are not exempt from these difficulties. Due to the lack of a more accurate metric, productivity is usually measure with LOC/man/month. We have already mentioned the possible misinterpretation that LOC can generate. A metric derived from LOC will also be affected by the same subjectivity factors.

II.2. Software Development Methodologies

II.2.1. Sequential Methodologies

a. The Boehm-Waterfall Model

The Waterfall Model was first introduced by Royce in 1970. In 1981, Boehm expanded the model adding additional steps[11]. This model described the software development process as a sequence of steps. The most common version includes seven non-parallel steps or phases, each of which includes validation against the previous one. If necessary, steps back to previous phases can be done (see Fig. 3).

[11] Blanchard Benjamin S., Fabrycky Wolter J., "Systems Engineering and Analysis", Third Edition, Prentice-Hall, 1998: p. 31.

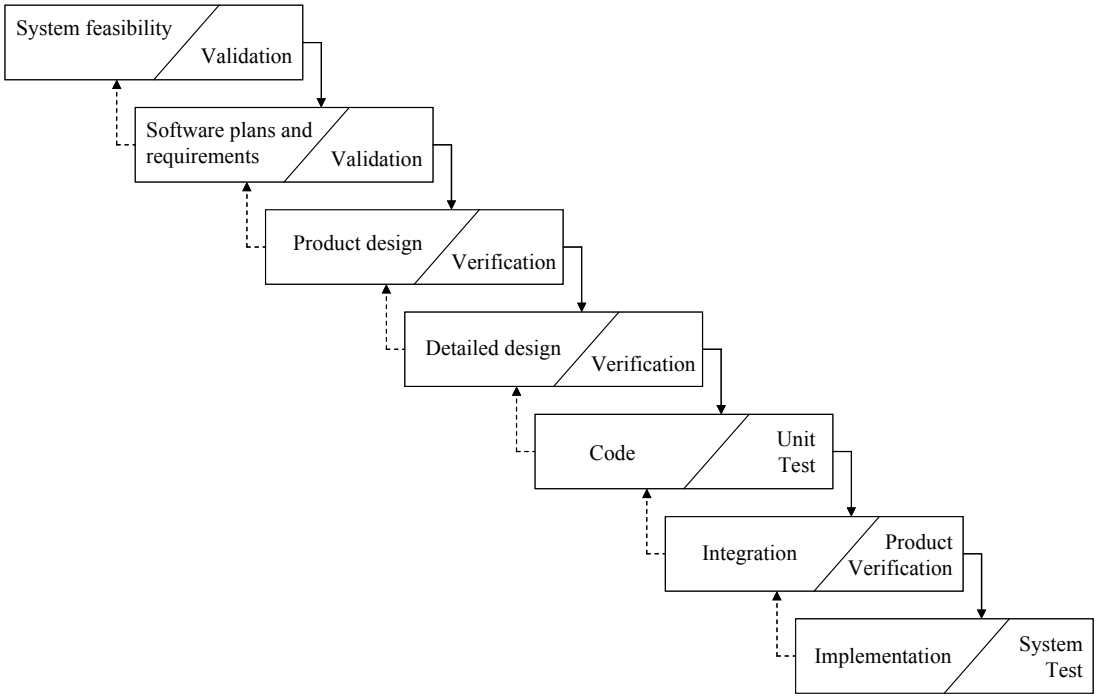

Figure 3. Waterfall Model. Source: Adapted from "Software Risk Management" (Boehm, 1989).

The Waterfall Model is a document-driven approach; communication strongly relies on the quantity and quality of the documents generated in each phase[12].

Over the years, this model has captured great attention in the software industry and become one of the most widely used models, especially in large government systems.

Some strengths of this model are:

- It was one of the first software engineering models
- It helped to develop a culture of thinking before coding
- It is easy to understand and adopt

Some of the main problems attributed with this model are:

[12] McConnell Steve, "Rapid Development", Microsoft, 1996: pp. 136-139.

- Changes in requirements have a great impact on the original schedule, the model forces to define requirements thoroughly during the System Requirements Definition Stage.
- Validation is not enough. Errors can escape this process and be found in further stages. In general, most of them are identified late in the project during the System Testing stage. To introduce changes at this point not only has higher costs but it can even be unfeasible.
- Feedback to previous stages is not easily introduced. In general, potential improvements would be included in future versions.

II.2.2. Iterative Methodologies

b. The Boehm-Spiral Model

The Spiral Model was developed by Boehm in 1986. This model leads the software development through a series of cycles or loops, each of which can be described as a reduced Waterfall model. The first cycle starts with an outlining of the objectives and an assessment of risk in meeting the objectives. The following cycles use feedback from previous stages to increase the level of detail and accuracy of the prospective system's objectives, constraints, and alternatives[13] (See Fig. 4).

Prototyping is needed for this model. A prototype is a reduced version of the system that is being built. They help to reduce the level of abstraction and improve the level of communication between users and developers. Prototypes are built and revised at the end of each cycle.

A recognized problem with this model has been the lack of guidance to increase the level of detail and accuracy after each cycle.

[13] McConnell Steve, "Rapid Development", Microsoft, 1996: pp. 141-143.

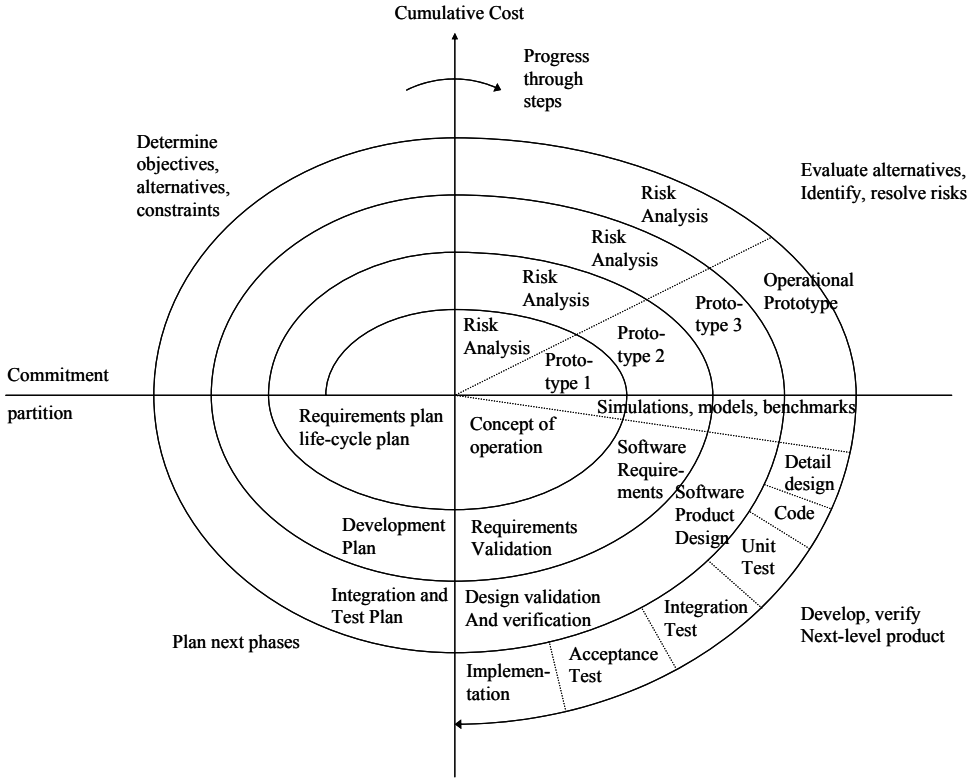

Figure 4. Spiral Model. Source: Adapted from "Software Risk Management" (Boehm,1989)

c. NGPM: A Win-Win approach to the Spiral Method[14] [15].

In 1994, Boehm and Bose introduced an extension of the Spiral model using the Theory Win-Win: the Next Generation Process Model (NGPM). This model allows stakeholders to impose heterogeneous constraints called win conditions. Theory W is then applied to manage individual concerns of the stakeholders and search for win-win solutions.

NGPM add two new sectors in each spiral cycle: "Identify Next-Level Stakeholders" and "Identify Stakeholders' Win Conditions" (see Fig. 5). It also adds a new a "Reconcile Win Conditions" task in the third sector.

[14] Boehm Barry W., Bose Prasanta, "A Collaborative Spiral Software Process Model Based on Theory W", USC Center for Software Engineering, University of Southern California, 1994: pp 1-2.

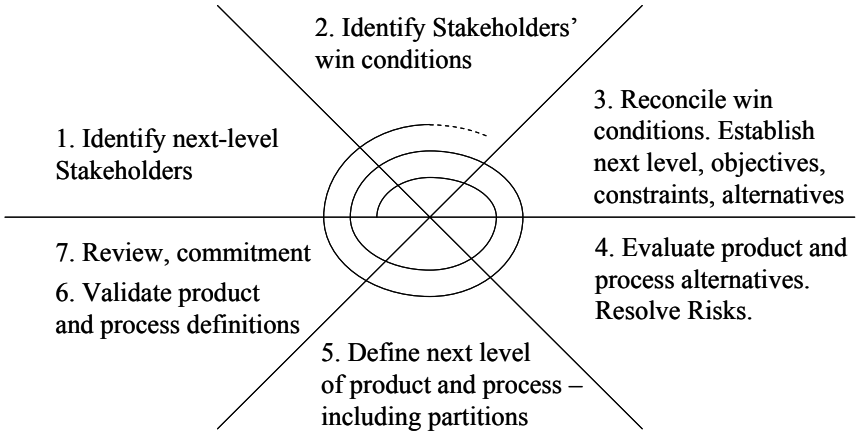

Figure 5. Spiral Win Win. Source: Adapted from "A Collaborative Spiral Software Process Model Based on Theory W" (Boehm, Bose, 1994)

d. Rapid Application Development

Rapid Application Development (RAD) appeared in the mid 1980's. It is a systems development method that arose in response to business and development uncertainty in the commercial information systems engineering domain[16].

RAD can be described as a response to two types of uncertainty: that of the business environment and that introduced by the development process. To address these issues, this methodology suggests: use of automated tools instead of manual code to increase productivity, people with knowledge of the business environment and communication skills should be involved in order to maximize feedback from users, and focus on development instead of analysis and design.

In the 1980's several companies released tools to implement the RAD methodology.

---

[15] Beynon-Davies P., Holmes S., "Integrating rapid application development and participatory design", IEE Proceedings Software, Vol. 145, No. 4, August 1998: pp 105-112.
[16] Really John P., Carmel Erran, "Does RAD Live Up to the Hype?", IEEE Software, September 1995: pp. 24-26.

e. Agile Software Development[17] [18]

In February 2001, a group of software engineers working in alternative development methodologies signed the so-called manifesto for agile software. In this document, they list a set of principles explaining their thoughts regarding the software development process. Business and technology have become turbulent, they said, and we need to learn how respond rapidly to the changes.

Unlike traditional approaches that focus on processes, documents, task distribution and development phases, the agile manifesto focuses on individuals, working software, customer collaboration and responsiveness to changes according to a plan. Agile software recognizes the importance of conformance to original plans but they claim satisfying customers at the time of delivery is most important.

The Agile manifesto states that using short iterations and working together with customers achieves better communication, maneuverability, speed and cost savings.

Among the most prominent Agile Methodologies that can be found are: extreme programming (XP)[19], Crystal Method[20], Dynamic Systems Development Methodology (DSDM)[21], and Adaptive Software Development (ASD)[22].

[17] Highsmith Jim, Cockburn Alistair, "Agile Software: The Business of Innovation", Software Management, September 2001: pp. 120-122.
[18] Highsmith Jim, Cockburn Alistair, "Agile Software Development: The people factor", Software Management, November 2001: pp. 131-133.
[19] See Beck Kent, "Extreme Programming explained", Addison-Wesley, 2000
[20] See Crystal Web Site (http://alistair.cockburn.us/crystal/index.html/)
[21] See DSDM Web Site (http://www.dsdm.org/)
[22] See Highsmith, James A., "Adaptive Software Development",Dorset House Publishing, 2000

Chapter III: Research Methodology

III.1. Comparative Analysis by phase

To understand the differences of the two approaches of software management a comparative analysis by phase will be developed. This analysis will identify the main features of each methodology and will contrast and highlight those aspects that differ the most. Whenever possible, quantitative data based on published studies is provided to illustrate these differences.

III.2. Interviews

As a complement of the comparative analysis, interviews of project managers working in different companies in Peru were conducted to understand what type of methodologies are in use and what the most relevant aspects were found in developing small and medium size software applications. These interviews were focused in three aspects: most relevant features of the small and medium sizes projects, main problems associated with software development, and effectiveness of formal methodologies.

The results of these interviews were also used as inputs in the development of the system dynamics model.

III.3. System dynamics model

As a complement to the comparative analysis and the interviews, a system dynamics model implementing the most relevant features of both approaches has been developed. This model uses a simplified version of the Waterfall Model to represent a traditional sequential approach. To represent the iterative approach this model uses a hybrid version based on extreme programming and agile methods. These two methodologies were selected because there is enough literature about them and because, arguably, they represent the most opposite approach to the traditional waterfall model.

Chapter IV: Analysis

This chapter discusses the most significant differences between sequential and iterative methodologies described in the previous chapter. It also explains the characteristics of the survey conducted to identify the key elements in the development of software that will be used to build the system dynamics model.

IV.1. Comparative Analysis

This part is organized into five sections each one describing a particular phase of the software development lifecycle. Goals, activities, and characteristics considering the points of views of sequential and iterative methodologies are discussed for each phase. So as to clarify the context, comments associated to the iterative approach are presented in *italic*.

IV.1.1. Definition Phase

The goal of this phase is to develop an initial understanding of the project. Based on this understanding a first estimation about the time and cost can be made.

The first phase of Waterfall Model, according to Boehm, is called "System Feasibility"[23] and it considers the development of a proposal with an initial estimation of the project. As part of the proposal, an initial project plan should also be delivered. Although there is always pressure for a precise estimation, project's features at this point are commonly vague and dynamic, and therefore estimations should be treated carefully. Authors have different opinions regarding how inaccurate these estimations can be. Rakos, for example, mentions

[23] Barry W. Boehm, "Software Risk Management", IEEE Computer Society Press, 1989: p. 27

studies done at NASA, DEC and TRW showing that "an estimate done at this point is 50% or 100% inaccurate"[24]. McConnell has a less optimistic view and suggests larger deviations at the beginning of the project (see Fig. 6)[25].

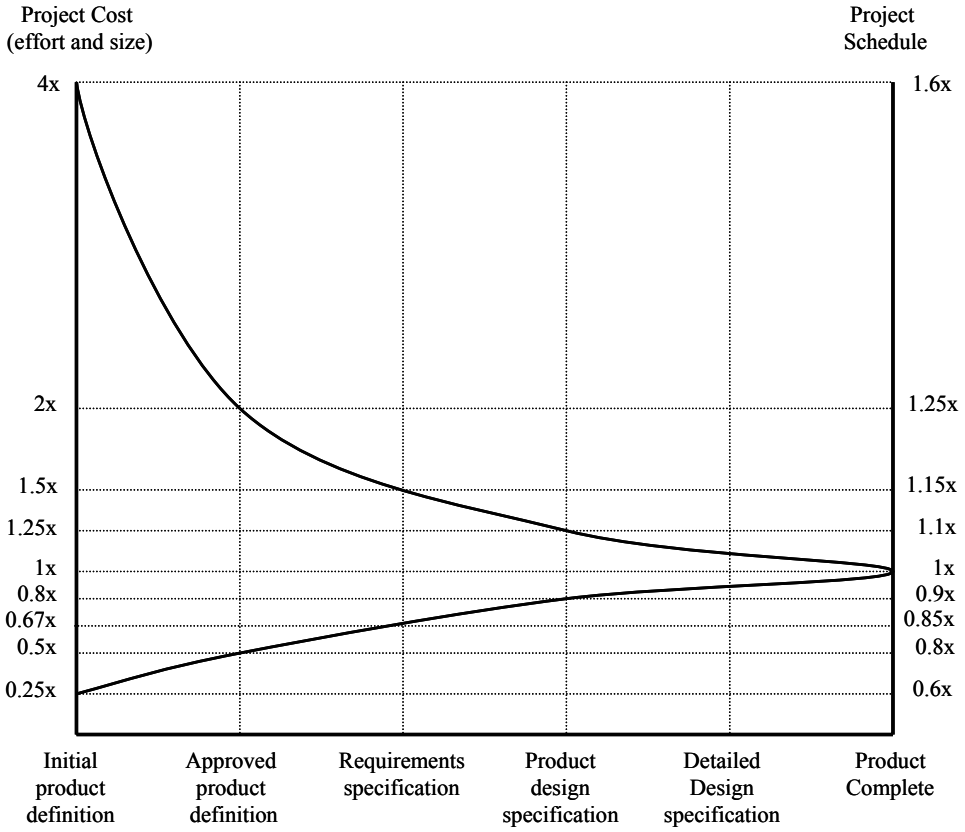

Figure 6. Estimate-convergence graph. Source: Adapted from "Rapid Development" (McConnell, 1996)

Another component that needs to be analyzed during this phase is the risk associated with the project. Thus, a list of potential risks including aspects such as technology, finance, resources, and schedule is developed.

[24] Rakos John J., "Software project management for small to medium sized projects", Prentice-Hall, 1990: p 128.
[25] McConnell Steve, "Rapid Development", Microsoft, 1996: p. 168.

The Spiral Model, as defined by Boehm, can be depicted as a sequence of several waterfall models growing in scope after each iteration[26]. Thus, each cycle begins with a definition of objectives, alternatives and constraints. The next step includes an analysis of alternatives and risks. The order and scope of each cycle is defined by the priority assigned to the remaining risks. In terms of scope, in a typical Spiral Model the first iteration could be compared to the first phase of a Waterfall Model.

Most recent iterative methodologies, such as agile methodologies an extreme programming, focus their attention on how best practices can be improved and adopted to achieve flexibility, quality, and productivity. Although a definition phase is not incompatible with these approaches, a more flexible working plan and scope should be considered. Extreme programming, for example, suggests breaking down a project into a series of small releases that are easier to plan and track[27]. A project using this methodology could sacrifice part of the remaining scope in order to keep the initial schedule. This can be achieved because the system is constantly under verification and, therefore, ready for the final testing phase. More differences between these methodologies and the typical Spiral Model are discussed in the next phases.

IV.1.2. Requirements

The goal of this phase is to define with a greater level of detail the functionality of the system. Some documents that can be considered in this phase are the functional specification, the analysis proposal and the top level design[28].

This is one of the most critical and yet uncertain phases for the Waterfall model. In its original version, this model doesn't consider overlapping between phases and requires a fair and complete documentation before moving to the next phase. However, the level of understanding about the requirements and the alternatives of implementation cause the

[26] Barry W. Boehm, "Software Risk Management", IEEE Computer Society Press, 1989: p. 39-36.
[27] Beck Kent, "Extreme Programming explained", Addison-Wesley, 2000: p. 56.

generation of ambiguous specification documents which, in turn, generate large quantities of code likely to be changed or eliminated. In 1988, Boehm recognized that because of its rigidity this model might not work well for some classes of software such as end-user applications[29].

A natural response to this problem is to relax the separation between the phases of the waterfall model. In practice, the distribution of activities within a phase is not absolutely even. Towards the end of a phase the number of tasks gets smaller and phase overlapping might help to reduce idle time. This approach, known as the Sashimi Model[30], is not exempt of problems. A common problem associated with this approach is the ambiguity that emerges as result of the overlapping effect, which increases the difficulty to track the progress and manage the project.

Iterative methodologies show a more radical solution to the changing nature of the requirements. Agile methods and extreme programming emphasize the use of the code to achieve a good definition of the requirements. For these methodologies, the scope of this phase is much reduced. It is used to define the priorities of the features to implement in the current cycle. The priority of the requirements is then used to reduce the scope if necessary. Agile methodologies combine analysis and development and recommend the user to be an active member of the development team. Using this approach, before coding there is only a list of basic functionality to be implemented. The level of detail is incrementally refined by the user and the development team through constant feedback. Extreme programming, for example, recommends having daily development cycles with an executable version of the system at the end of each day.

Another aspect that is suggested by the extreme programming methodology is the use of storyboards as a tool to describe what the system should accomplish. These storyboards are short and simple business "stories" directly related to a specific functionality of the system

---

[28] Rakos John J., "Software project management for small to medium sized projects", 1990 Prentice-Hall: p.56-69.
[29] Barry W. Boehm, "Software Risk Management", IEEE Computer Society Press, 1989: p. 28.
[30] McConnell Steve, "Rapid Development", Microsoft, 1996: p. 143-145.

and are extensively used to test and verify the quality of the development. Storyboards are defined before the coding and guide the team through the development process.

IV.1.3. Design

The goal of this phase is to define the architecture of the system. According to the Waterfall Model, the design comprises two phases: product design and detailed design[31]. The product design identifies the top level components of the system and how they interact. The detailed design described the architecture of each component. This design is included in a document called the Design Specification. The Waterfall Model also suggests elaborating the Acceptance Test Plan during this phase. This document, whose development should be led by the user, explains what tests will be performed to validate that the system is working properly and according to the original specifications.

Boehm called the third round of a spiral model as the "top-level requirements specification". In a typical development, this phase could be compared to the design phase of the waterfall model. However, there is a significant difference. The spiral model focuses the organization and the scope of its cycles around the risks of the projects. Each round helps to address a set of risks and therefore decide what to do next. Using this approach, every project could have a different scope for every round[32]. A small project could have fewer iterations than a large one.

Agile software and extreme programming have a different approach to the design. The design and the coding are highly coupled tasks and, therefore, should be performed together. These methodologies claim that the user and the developer increase their understanding of the problem with each new iteration. To leverage this understanding, extreme programming

[31] Boehm, Barry W., "Software Risk Management", IEEE Computer Society Press, 1989: p. 27-28.
[32] An example of how this principle can be applied to elaborate a project plan can be found in the development of Microsoft Office 2000. This project was broken into major milestones each of them with a death-line. When a milestone was delayed, the scope in next stages was reduced so that the schedule could still be attained (Harvard Business School, 9-600-097, June, 2000).

recommends short development cycles of one to four weeks and continuous daily integration and unit tasks[33].

IV.1.4. Coding

The goal of the coding phase is to write the necessary code to implement the system specified in the design. Along with the code, other typical deliverables of this phase are the System's Guides for the user, the maintenance, and the operator). Coding is usually perceived as the easiest phase probably because the level of ambiguity decreases as more code is developed.

In his articles, Brooks mentioned studies that show that an experienced programmer can be up to 10 times more productive than non-experienced ones[34]. This significant difference might have been ameliorated by the evolution of programming languages but still remains as an important factor of success and needs to be considered. Because of this important difference experienced programmers are more expensive resources. Potential causes of delays must be identified in advance to avoid idle times.

In a typical waterfall model, coding starts after the design has been completed. However, due to the users' pressure for concrete results, to initiate coding tasks before design is completed is a common practice. If not properly managed this might become a risky practice because, in a sequential approach, early phases' definitions are more susceptible to changes and, hence, developing code until complete approval might require considerable rework. As in any other phase of the waterfall model, at the end of the coding phase validation tasks must be performed. These activities are called unit tests.

The spiral model is more flexible and coding can be introduced in any iteration. However, in a typical implementation, coding starts in the third or fourth iteration depending upon the size of the project. It can start with some type of prototyping and in the next iteration deliver

[33] Beck Kent, "Extreme Programming explained", Addison-Wesley, 2000: pp. 56, 59, 64, 133.
[34] Brooks Frederick, "The Mythical Man-Month", Addison Wesley Longman, 1995: p. 30.

the first integrated version of the system. Regarding the type of activities to be included in the coding phase this model is not significantly different from the waterfall model.

Proponents of agile methodologies believe that code is the only real deliverable of a software project and therefore it should be used to learn and gain insight from the beginning of the project. Analysis and design tasks should not be separated from coding but rather they all be performed together in small iterations.

Another difference brought by these methodologies is the role of testing. Unlike waterfall and spiral models that suggest to consider unit tests at the end of coding, these new approaches established that testing should be performed along with coding. Another important feature of coding in Extreme programming is pair programming, a technique that is supposed to increase the quality of the code. This technique allows two programmers to collaborate at one computer, typically one person using the keyboard and the other one using the mouse. Proponents of this technique claim that two people looking at the code increases the likelihood of finding mistakes. They also argue that it increases the creativity of the team and improve the learning experience[35].

Pair programming is a technique that addresses one aspect of coding and it is not exclusive of any methodology. It can also be applied in waterfall and spiral models.

IV.1.5. Testing

The goal of the testing phase is to validate that the system performs properly according to the initial specifications. Tests are conducted reproducing tasks that the users will do after the system is released. For large or complex programs it is suggested to develop small programs that test the system automatically.

[35] A complete description of this technique can be found in Williams Laurie, Kessler Robert, "Pair Programming illuminated", Addison-Wesley, 2002.

The waterfall model recommends performing validation tasks at the end of each phase. After tasks have been approved the next phase is not officially started. Although validation helps to identify possible errors, due to the first phases' ambiguity, definitions and requirements even in written documents might be understood in different ways, which makes the validation tasks more difficult. During the coding phase unit tests are performed. These tasks validate a specific component or function. The testing phase is the first time that the product is tested as a whole. Modified versions of the waterfall model suggest overlapping between phases. Using this approach, testing might overlap during the coding and, thus, rework might be discovered earlier.

The spiral model introduces validation activities after the third round along with the coding activities. Just as in the waterfall, the spiral model suggests to perform coding, unit tests, and integral tests separately. The difference with the waterfall model is the scope considered by each iteration, which may increase the ability to identify errors earlier.

Extreme programming recommends a completely different approach. Proponents of this methodology believe that testing is also a central activity of development and needs to be executed along with coding. Moreover, they say that testing cases and tools should be developed before the actual code. This helps the team to identify errors in earlier stages and, thus, reduce unnecessary rework.

The following phases after testing, such as acceptance, implementation, and maintenance, do not show significant differences between the iterative and sequential approaches.

IV.2. Interviews

The interviews were conducted with 12 project managers in 5 different companies in Peru. These project managers received a survey form including questions regarding their experience in software development. After forms were completed they were consolidated in a database. The complete set of questions and results is presented in Appendix A.

Results show that, on average, small and medium software projects in these companies have a duration of 7.4 months and required 54.2% more time to complete than originally estimated (See Table 1). Regarding the quality of these projects, only 56.2% were perceived as successful. However, success might have different interpretations. To clarify this concept the project manager were asked to define this concept.

	Total	Average	Standard deviation
Number projects develop in the last 2 years	66 projects	4.9 projects	2.3 projects
Initial estimated duration		4.8 months	1.6 months
Real duration		7.4 months	2.6 months
Testing phase duration		1.5 months	0.6 months
Development team size		7.7 people	3.2 people
% of successful projects		56.2%	29.9%

Table 1. Small and medium sized projects' characteristics in Peruvian companies.

Table 2 shows a distribution of the terms used by the project managers to define success. The three most frequently mentioned aspects were the users' requirements, cost and time. It is interesting to observe that important aspects such as system internal quality and the total value of ownership were rarely included in the definition of success.

A successful project is one that …	%
… satisfies the user's requirements	92%
… doesn't need budget extensions	58%
… ends on time	50%
… exceeds user's expectations	25%
… creates value to the company	25%
… ends reasonably on time (less than 10% delay)	17%
… has a long useful life	17%
… has an excellent technical design	8%

Table 2. Software project success' definitions

The second part of the interviews intended to gain more understanding of the common problems that software projects face. A list of possible problems was initially suggested and project managers were asked to rank these aspects according to their importance, which was defined as the level of impact these problems may have if occurred. Finally, relative weights

were assigned to consolidate the results. If a particular task was selected by all project managers as the most important it would receive a score equal to 100%.

Table 3 shows that the most important source of problems is the user specifications' ambiguity, a common characteristic associated with software that highlights the difficulties software engineers face to develop a common language with users. Unrealistic planning was selected as the second most important problem. It must be noticed that all the participants of these interviews acknowledged having trouble achieving initial estimations. In this regard, 75% of the participants identified coding as the phase most likely to be delayed. The third problem was the number of changes in the specifications after analysis phase was completed, which is closely related to the first problem. Changeability has been largely identified as one of the biggest challenges in developing software and it is been the inspiration for several new development methodologies. In fact, the agile manifesto includes as one of its principles the following "Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage"[36]. In other words, changes should not be considered a nuisance or risk but an opportunity.

Source of problems (sort by importance)		%
1	User specifications' ambiguity	89%
2	Unrealistic planning	80%
3	Changes in the specifications during planning	77%
4	Lack of technical experience	58%
5	Lack of a development methodology	58%
6	Poor risk management	52%
7	Parallel projects affecting team's productivity	48%
8	Low budgets	36%
9	Poor coordination to allocate resources	34%
10	Poor testing	28%
11	Lack of motivation	25%
12	Other technical problems	21%

Table 3. Most common problems sort by importance

Regarding the use of methodologies, a third of the participants said that they don't use any commercial methodology but rather one developed based on the good practices of the

[36] "Manifesto for Agile Software Development" web site (http://agilemanifesto.org/)

company. The rest of participants mentioned three methodologies: Capability Maturity Model (CMM), Project Management Office (PMO), and Microsoft Solutions Framework (MSF). All these methodologies are intended to improve organizational management practices and are, in theory, independent of whether the development is following a sequential or iterative approach. Finally, regarding the use of a sequential methodology or iterative, all the participants affirm to use waterfall methodologies based hybrids. Some of the more typical variations were phase overlapping and parallel sub-projects.

IV.3. Simulation Model

IV.3.1. Overview

This model simulates the development of a small sized[37] information system within a typical business organization. The model recreates this development considering two different development methodologies: the first is a sequential waterfall-based approach, and the second is an iterative-based approach that gathers elements from agile and extreme programming methodologies.

The model defines the projects a static set of tasks to be worked through different development phases. Each phase has a different development rate. In the sequential approach the first three phases have two distinctive parts: development and verification. During development tasks are worked but no verified. Verification starts after all tasks have been worked. During this part tasks that need to be reworked are detected and sent back, the rest are approved and sent forward to the next phase. The next phase doesn't start until all the tasks have been verified and approved. However, because of the lack of insight and verification rigor in the initial stages of a project, a subset of tasks that need rework are not identified as deficient during verification and are mistakenly approved. In the next phase

[37] The definition of small sized used in this section refers to the average duration of projects calculated from the interview answers: 7 men x 5 months ≈ 3 man years.

these tasks will generate more rework. In the last phase, testing, the verification accuracy increases and most of these tasks are finally identified and sent back to be fixed.

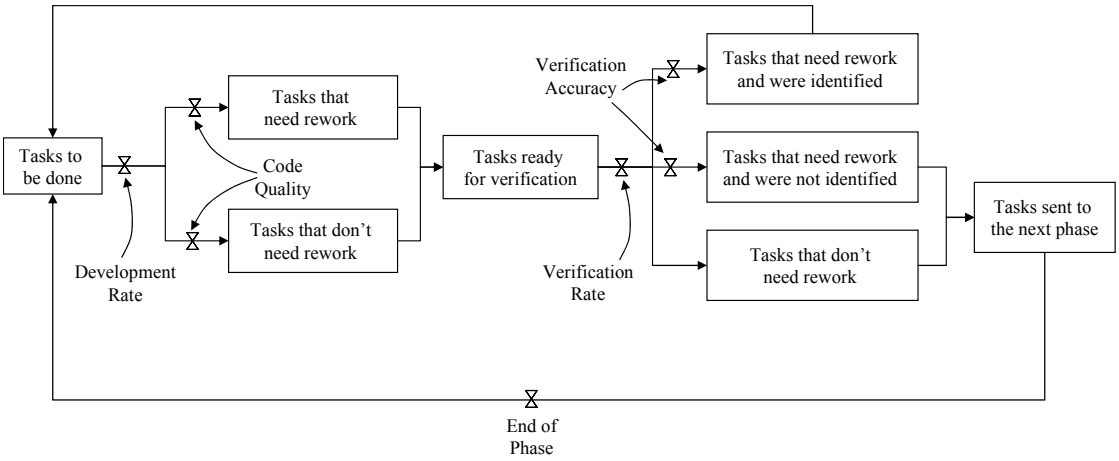

Figure 7. High level view of the Model.

In the iterative approach, analysis, design, and coding are grouped into one big first phase that includes several small cycles of validation. The size of each cycle can change between methodologies. A spiral model could have iterations of many weeks. Some more recent approaches such as extreme programming recommend performing integration and validation tasks on daily basis. For the model this period is adjustable to any value.

IV.3.2. Description of the Model

a. The phases considered.

The model represents the evolution of a project starting from the analysis phase thru the testing phase. The feasibility phase, as described in the waterfall model and spiral model, was not considered in this model. It was excluded because the nature of this phase is significantly different from the rest of the phases: it involves fewer people, it is not immediately followed by the next phases of development (there are usually preparation activities before the development actually starts) and it can be equally applied to sequential

and iterative methodologies. To simplify the model only four phases has been considered for the sequential approach: analysis, design, coding and testing[38]. The iterative model was implemented as a subset of the sequential approach, considering only two phases. The first phase comprises a series of iterations including analysis, design and coding tasks. The second phase is testing. These assumptions allow to better extract the important differences of these approaches.

Appendix E includes a more detailed view of testing phase at one of the companies that participated in the interviews. A slightly different model focusing only in testing is used to analyze how testing configuration might be adjusted to get better results.

b. The size of the project.

The simulation considered a small project with 90 function-points. A function-point is a unit that, unlike the traditional lines of code, considers different elements to estimate the size of a program such as user-interface components, applications interface components, and data storage components (files, tables, etc.). Because of this feature function-points are more suitable for projects with a high number of database components such as an information system. Using the factors provided by Jones[39] a project with 40 components may generate between 90 and 123 function-points depending on its complexity. This measure is then used to estimate the necessary schedule to complete a project of this size. Considering an average quality level, a 40 components project may require approximately 0.43^{102}=7.3 months to be completed. This corresponds to the average actual duration of projects identified in the interviews.

c. The size of the phases

The relative effort associated with a phase varies upon several elements such as the size and type of project, the development methodology or the experience of the development team.

[38] In addition to these phases, Boehm's Waterfall Model considers a Feasibility phase, a Preliminary Design Phase and an Implementation Phase.

For this model, we are considering the relative sizes for a small project suggested by McConnell[40]. For the iterative approach the first phase has a size equivalent to the first three phases of the sequential approach.

d. The rework generation

Rework is defined in this model as function of the level of understanding the team and the user have about the application they are developing. This level of understanding or Insight increases as the users and the development team move forward through the phases. A greater Insight generates a smaller level of rework. Previous models, such as Madnick and Tarek's[41], have studied the impact on productivity and quality due to experience and learning. This level of understanding is qualified in the model described here as the "insight". To estimate the appropriate level of insight, we considered an initial level for each phase and also a curve describing how the insight evolves. The model described insight as a function of the % of tasks verified. It assumes that understanding is achieved after verifying work in progress and how well it satisfies the systems specifications. To estimate the initial insight of each phase the average uncertainty suggested by McConnell for each phase[42] was used. Then, three different evolution curves are considered in the model: the first assumes that there is a good clarity at the beginning of each level and therefore not much insight is gain with the first stages of the progress, the second curve assumes a linear increase of the insight, and the third curve assumes that much insight can be gained from the start of each phase. The insight is used to estimate the percentage of rework that is generated at any point in time.

e. The resources

---

[39] Jones Capers, "Applied software measurement: assuring productivity and quality", McGraw-Hill 1991

[40] McConnell Steve, "Code complete: a practical handbook of software construction", Microsoft Press, 1993: p 522.

[41] Stuart Madnick, Tarek K. Abdel-Hamid, "The dynamics of software project scheduling: a system dynamic perspective", Center for Information Systems Research, Alfred P. Sloan School of Management, 1982: p. 83, 98-99.

[42] McConnell Steve, "Rapid Development", Microsoft, 1996: p. 122.

The model assumes a stable team during the entire project. The addition of resources to the project during the development is not considered in this model because the effects of this practice have been largely studied before and because, being a small project, the learning time for a new resource to become productive invalidates hiring as a common practice for small projects. This model considers a different team for validation and testing. Therefore, development and validation can be performed in parallel. However, since validation and testing frequently needs the participation of the development team to solve questions and show the users the results, the model considers a percentage that is subtracted from the development rate when validation or testing is being conducted in parallel.

f. The productivity

This model considers an increase in the productivity due to learning. For this project a 25% increase of productivity associated with this factor was considered. This percentage was suggested by Madnick and Taruk based on IBM studies[43]. Other external factors such as motivation and pressure are not being considered because their impact on a small project is less significant.

g. Other assumptions

To build a simple and accurate model was a premise of this work. Simplicity is important because the development of a model is by itself an iterative job that needs many cycles until it behaves properly. Initial results usually point out opportunities for improvement and simple models facilitate the identification of mistakes and the introduction of improvements. Simplicity should not be confused with lack of accuracy. Accuracy, defined as the conformity to reality, is achieved identifying the most important elements of the process and understating the type of relations they have. Although this model doesn't reflect a specific project in particular, accuracy can be confirmed by comparing the initial results with those suggested by the literature and data obtained from the interviews.

[43] Stuart Madnick, Tarek K. Abdel-Hamid, "The dynamics of software project scheduling: a system dynamic perspective", Center for Information Systems Research, Alfred P. Sloan School of Management, 1982: p. 83.

In order to develop a simple model that focuses on the more important aspects of the methodologies object of this study, some assumptions were employed. These assumptions are as follows:

- Constant number of tasks. Although a constant number of tasks is not a frequent scenario, this assumption was made because this is a management issue that is independent of the methodology approach used and, therefore, out of the scope of this study.
- No delay pressure or other factors affecting the motivation are considered. Unlike the previous assumption, changes in the motivation can dramatically impact the development speed and quality of a project. They can also behave differently in an iterative or sequential approach. However, since the model represents small projects it is reasonable to assume that the impact of changes on the motivation is not significant.
- Tasks that need rework are only reworked in the current phase. In theory, both iterative and sequential approaches contemplate the possibility of sending a task back to a previous phase. Several authors have studied how the cost to fix a mistake increases as the project moves forward. Conte, Dunsmore, and Shen mentions studies made at IBM, GTE, and TRW showing that "… an error introduced during the requirements phase, but not discovered until maintenance, can be as much as 100 times more than that of fixing the error during the early development phases …"[44]. Likewise, Madnick and Tarek found that as the error density goes down the more expensive it becomes to detect and correct errors[45]. This increasing cost is caused by the overhead time to fix tasks from previous phases and by the additional rework that tasks with errors generate. Although to capture this effect would be beneficial to increase the accuracy of the model, it would require the creation of a specific set of levels for each phase which, in turn, would increase dramatically the number of elements and relations of the model. For that reason, with the exception of the testing, this model considers that rework is only done in the current phase. However, the

[44] Conte S. D., Dunsmore H. E., and Shen V.Y., "Software Engineering Metrics and Models", Benjamin/Cummings, 1986: p. 7.

model does keep track of the tasks mistakenly approved in the previous phase and use it as a variable to calculate the quality of the work done in the next phase.

h. Model Parameters

- *Number of tasks*. Number of tasks or function-points that will be developed. This value is set to 102.
- *Normal Development Rate*. Number of tasks that can be developed in one day by the whole team. To complete a 102 project function-point project in 7.3 months it will be necessary a net development rate equal to 102 function-points /7.3 months/ 22 days = 0.63 function-points/day. However, this rate must take into account non-productive time associated with rework and support in validation tasks. Assuming a 50% of time spent on those activities the development rate to finish in 7.3 months would be $0.63/0.5 \approx 1.2$ function-point per day.
- *Normal Verification Rate*. Number of tasks that can be verified in one day by the whole team. This model assumes that verification activities can be performed at twice the speed of development.
- *Flag Iterative or Sequential*. When it is set to 0 the model simulates a sequential type of project. A value equal to 1 forces the model to simulate an iterative type of project.
- *Verification Period*. When the model is simulating an iterative type of project this parameter set the number of days elapsed between two verification cycles.
- *Phase Size Table*. It indicates the percentage of effort that each phase of development represents. The values were adapted from McConnell[46]. The values for the sequential approach are as follows: Analysis or Phase 0 = 0.1, Design or Phase 1=0.2, Coding or Phase 2=0.45, and Testing or Phase 3=0.25. For the iterative approach the values are Development or Phase 0 = 0.75 and Integration and Testing = 0.25.

---

[45] Stuart Madnick, Tarek K. Abdel-Hamid, "The dynamics of software project scheduling: a system dynamic perspective", Center for Information Systems Research, Alfred P. Sloan School of Management, 1982: p. 105-106.
[46] McConnell Steve, "Rapid Development", Microsoft, 1996: p. 122.

- *Insight per phase*. It reflects the level of understating the users and the development team have about the project. This level defines the % of tasks that will need to be reworked at any point in time during the project. In this model the insight is a value that increases in each phase as a function of the % of tasks that has been verified within in each phase. The initial values of insight were based on the inaccuracy suggested by McConnell[47] at each phase of the project. The values are as following: Phase 0 = 0.25, Phase 1= 0.5, Phase 2 = 0.7, Phase 3 = 0.85.
- *Insight development*. This curve represents the insight evolution within each development phase. Three curves were proposed to represent respectively slow, average, and fast insight development.
- *Verification accuracy*. It indicates the probability that a task with errors is identified during verification activities and sent back to development. This likelihood increases with each phase. For the sequential approach the values are Phase 0 = 0.5, Phase 1=0.55, Phase 2=0.65, and Phase 3=0.90. The iterative approach uses the values of Phase 2 and Phase 3. These values are not based on any previous study and are proposed here based on my own experience.
- *% dedicated to verify*. It measures the percentage of time the development team is dedicated to support verification activities, i.e. interaction with the testing team.

i. Model Stocks[48]

The following paragraphs provide a brief description of the main stocks of the model and their inputs and outputs.

- *Tasks to be done*. This stock stores the number of tasks that need to be worked in a phase. At the beginning of each phase the value of this stock is equal to the *Number of*

[47] McConnell Steve, "Rapid Development", Microsoft, 1996: p. 168.
[48] Stocks (also known as Entering Levels, State Variables, or Accumulations) and Flows are the basic elements used to build the principle of accumulation in system dynamics. A stock can be depicted as a bathtub. A flow can be thought of as a pipe and faucet assembly that either fills-up or drains the bathtub. A complete explanation of System Dynamics can be found in Forrester, Jay W., "Industrial Dynamics", Pegasus Communications, 1961

tasks. At the end of each stock the value of this stock is zero. During a phase tasks are worked according to the development rate. Those with errors detected are sent back to this stock.

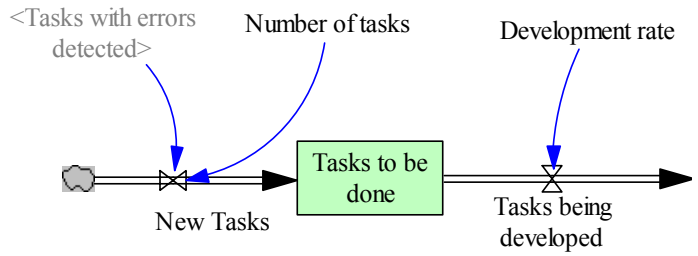

Figure 8. Tasks to be done.

- *Tasks ready for verification*. After development tasks are completed they are immediately sent for verification. This stock stores the number of tasks waiting to be verified. The *Verification Rate* parameter defines how many tasks are verified each day and the variable *Time to verify?* defines how often. In the sequential approach tasks wait until the end of the phase to be verified. In the iterative approach tasks are constantly being verified. At the beginning and at the end of each phase the value of this stock is equal to zero.

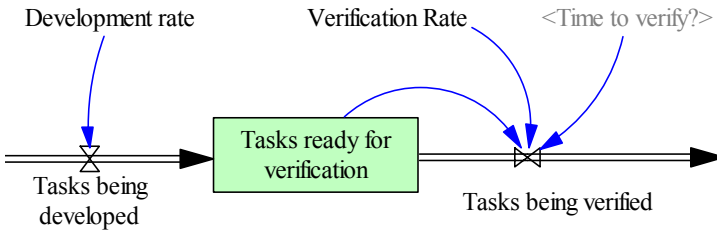

Figure 9. Tasks ready for verification.

- *Tasks that need rework before verification*. This stock stores the number of tasks that have been worked and need rework. Its input, *Tasks being developed with errors*, is a percentage of *Tasks being developed*. This percentage changes over time and depends on the insight and the quality of previous phases. All tasks needing rework that were

mistakenly approved during verification will come back to this stock in the subsequent phase.

- *Tasks that need rework after verification*. This stock stores the false negatives, that is the number of tasks with errors not detected during verification.
- *Tasks that were reworked*. This stock stores the number of tasks with errors detected and sent back to development again.

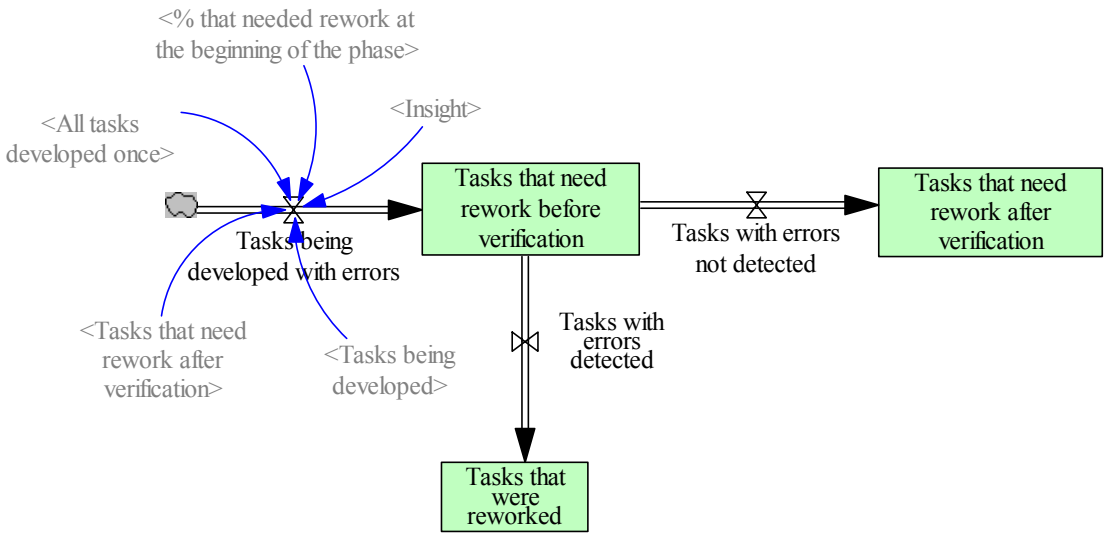

Figure 10. Task that need rework.

- *Tasks that don't need rework before verification*. This stock stores the number of tasks that have been worked and don't need rework. Its input, Tasks being developed without errors is equal to the Tasks being developed minus Tasks being developed with errors.
- *Tasks that don't need rework after verification.* This stock stores the number of tasks without errors that were approved during verification.

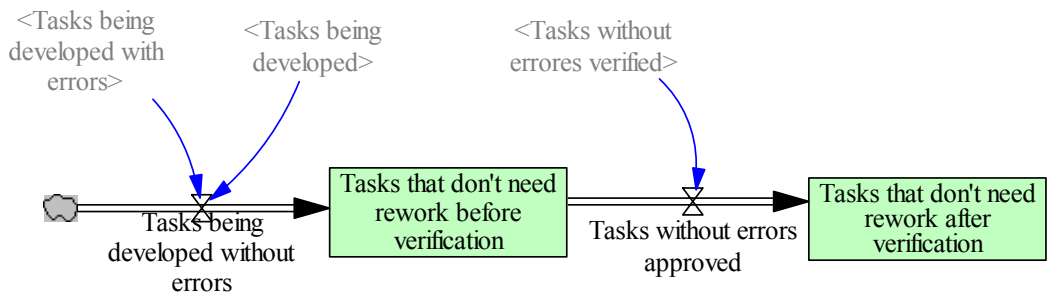

Figure 11. Tasks that don't need rework.

j. Main Flows

- *Tasks being developed*. This flow measures how many tasks are develop a day. When *Tasks to be done* has a positive value, this flow is equal to the variable *Development Rate* (see Main Variables).

- *Tasks being developed with errors*. This flow is a fraction of *Tasks being developed*. Its formula is:

*Tasks being developed with errors = Tasks being developed * (1-Insight)*

At the beginning of each phase and before all tasks have been sent at least once to verification the formula also includes the quality of the previous phase. In other words the quality of the code at the beginning of a phase cannot be better than the quality of the code at the end of the previous phase. The quality starts to improve when verification activities starts. The formula is:

```
Tasks being developed with errors =
        Tasks being developed *
        (
          % of tasks that need rework at the beginning of the phase +
          (1-Insight(Phase)) * (1-% of tasks that need rework at the beginning of
          the  phase)
        )
```

- *Tasks being developed without errors*. The value of this variable is calculated subtracting the tasks with error from Task being developed. Its formula is:

Tasks being developed without errors =
Tasks being developed - Tasks being developed with errors

- *Tasks being verified*. It is a fraction of the Normal Verification Rate. Its formula is:

Tasks being verified =
Normal Verification Rate / (Relative Size (Phase))

- *Tasks with errors detected*. It is fraction of *Tasks being verified*. Its formula is

Tasks with errors detected =
*Tasks being verified * (% that need rework) * Verification Accuracy (Phase)*

- *Tasks with errors not detected*. It is fraction of *Tasks being verified*. Its formula is

Tasks with errors not detected =
*Tasks being verified * (% that need rework) **
(1-Verification Accuracy (Phase))

- *Tasks without errors approved*. It is fraction of Tasks being verified. Its formula is

Tasks with errors detected =
*Tasks being verified * (1-% that need rework)*

k. Main variables

- *Insight*. The insight measures the level of understanding the development team and the user have about the application they are building. This variable takes its value form the variable *Insight per phase* table and it increases as a function of the verification tasks. Each phase has an initial level of insight and it grows as a function of the % of tasks verified. *Insight type* indicates whether the development of the insight is slow, average of fast. The Insight is used to calculate the rework generation.

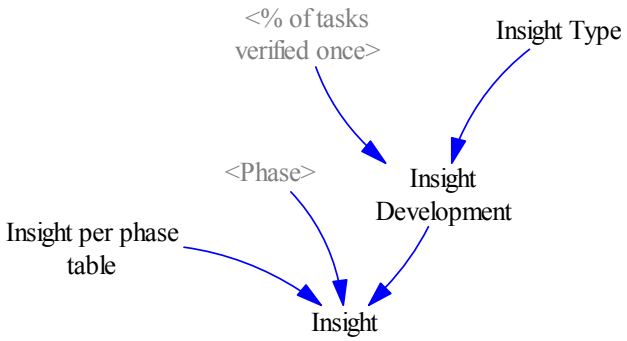

Figure 12. Insight

- *Productivity Factor*. In their model, and based on Aron's studies[49], Madnick and Tarek[50] depicted improvement in productivity due to learning as an S-shaped curve that goes up to 25% at the end of the project. This model uses the same approach and implements productivity as a linear function of the percentage of worked tasks. Since different project phases have different type of tasks and, therefore, experience gained in one phase has little impact on other phases, this model calculates productivity evolution for each phase independently. In this model productivity doesn't impact the quality of development it only speeds up the pace of development.

- *% that need rework*. It is the % of tasks that need rework. Its formula is:

  % that need rework =
  Tasks that need rework before verification /
  (Tasks that need rework before verification +
  Tasks that don't need rework before verification)

- *Development Rate*. This variable measures the number of tasks that can be developed in a day. To calculate this value, the *Normal Development Rate* is divided by the relative size of each phase. Another element that affects this value is the percentage

[49] Aron J. D., "Estimating resources for Large Programming Systems", Litton Educational Publishing, Inc., 1976
[50] Madnick Stuart, Abdel-Hamid Tarek K., "The dynamics of software project scheduling: a system dynamic perspective", Center for Information Systems Research, Alfred P. Sloan School of Management, 1982: p 83.

dedicated to support verification activities. This percentage is only considered when tasks are being verified.

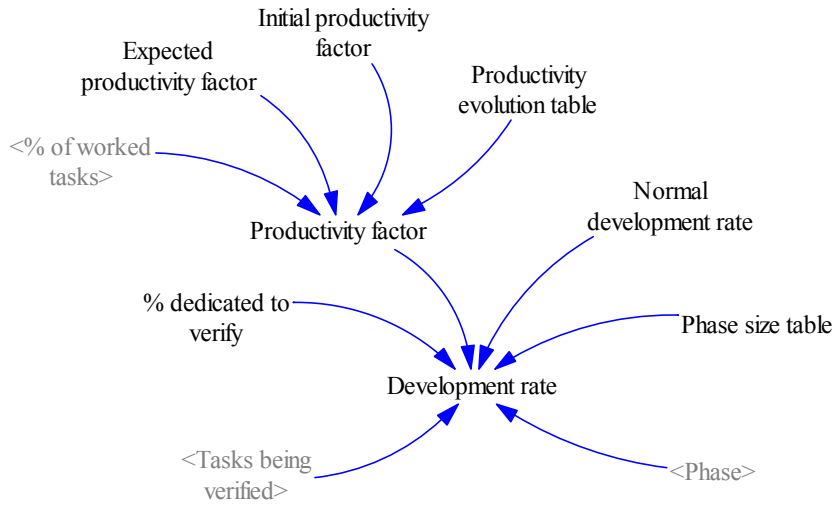

Figure 13. Development rate.

Table 4 shows a consolidated overview of the main parameters of the model.

Parameters	Sequential	Iterative
# Tasks	102	102
# of Phases	4	2
Relative Size of Phases		
Analysis	10%	75%
Design	20%	
Coding	45%	
Testing	25%	25%
Verification Accuracy		
Analysis	50%	65%
Design	55%	
Coding	65%	
Testing	90%	90%
Verification Period	After tasks has been developed	Daily process
Initial Insight per phase		
Analysis	25%	25%
Design	50%	
Coding	70%	
Testing	85%	85%

Table 4. Overview of parameter settings

Chapter V: Results

This section shows the results of the model using the default values for the parameters. Fig. 14 and Table 5 show the time of completion and the number of errors that were not detected at the end of the project.

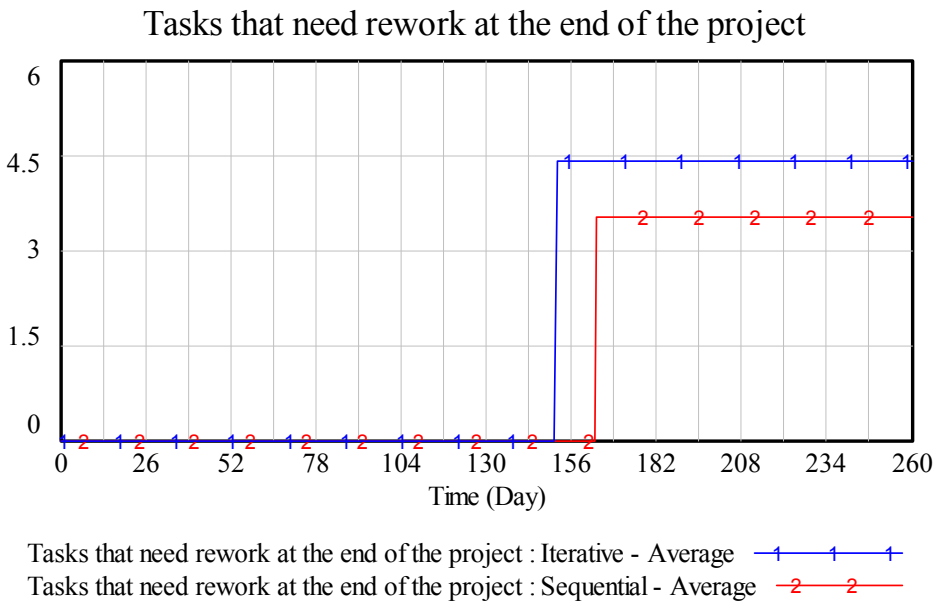

Figure 14. Default scenario results.

	Iterative	Sequential
Number of days to completion	152 (5% smaller than expected)	164 (2% larger than expected)
Tasks with error after completion	4.39 (4.3% of 102)	3.52 (3.4% of 102)

Table 5 Default scenario results.

Although the iterative approach finishes the project earlier, the quality of the final product, in terms of undetected errors, is lower.

A good starting point to understand these results and how different the two approaches are is to analyze the distribution of tasks by stages of work. A task can only be in one of these possible stages: waiting for development, waiting for verification, or waiting for the next phase. The stocks that store this information are respectively: *Tasks to be done*, *Tasks ready for verification*, and *Tasks ready for next phase*. Figures 15 and 16 show this distribution for both approaches. The sequential approach shows a similar performance for the first three phases. At the beginning of each phase tasks are only developed. Verification starts after all tasks have been developed and then some tasks needing rework are detected. It is observed that time spent doing verification and rework tasks accounts for more than the half of the first three phases. The last phase, testing, shows a different type of distribution because it doesn't start with any development and tasks are immediately sent to verification. The iterative simulation shows only two phases. The first phase shows some differences when compared to the first phases of the sequential approach. Some of these differences are: the decreasing number of *Tasks to be done*, the small number of *Tasks ready for verification*, and the small but steady improvement in the rate at which the number of *Tasks ready to the next phase* increases. Another difference displayed in the iterative approach is the oscillation in the number of tasks to be done. This pattern is caused by the % of tasks with errors that are discovered during the verification tasks. The length of these oscillations corresponds to the size of the verification cycles.

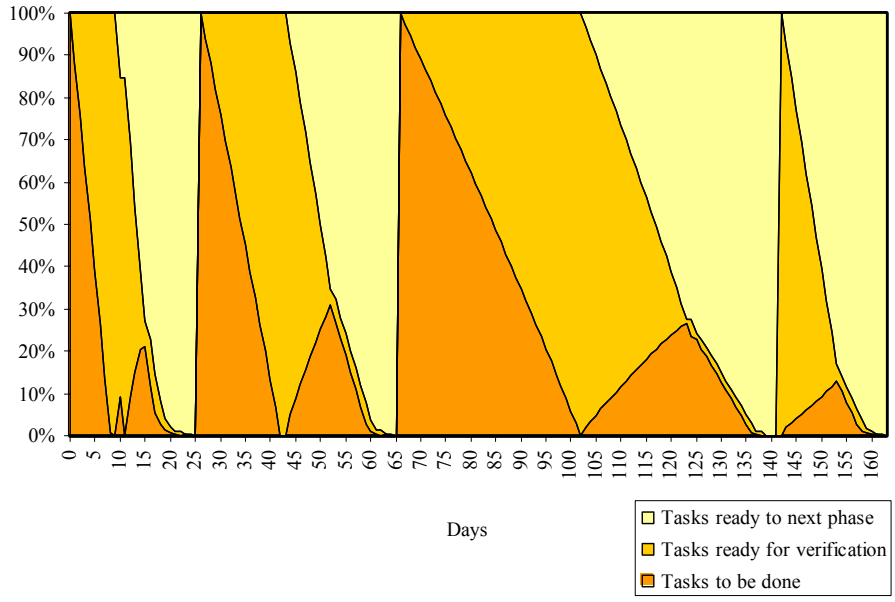

Figure 15. Task distribution by development stage (sequential approach)

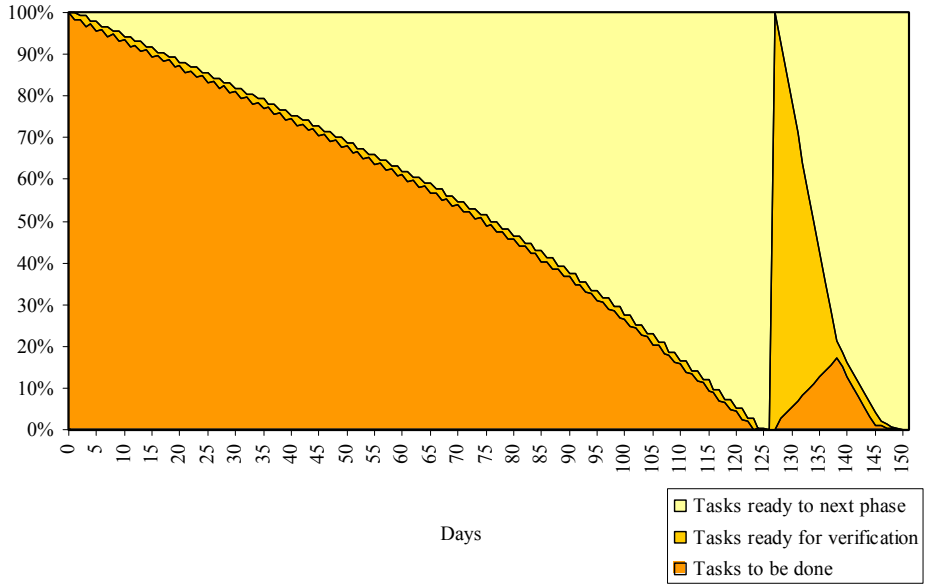

Figure 16. Task distribution by development stage (iterative approach)

The following paragraphs analyze some variables of the model to understand their impact on the results. The first stock to be studied is "% of work accomplished". Fig. 17 shows that, with a minor exception at the beginning of the project, work was accomplished at a faster pace in the iterative model, which gives an initial indication of why this approach ends first.

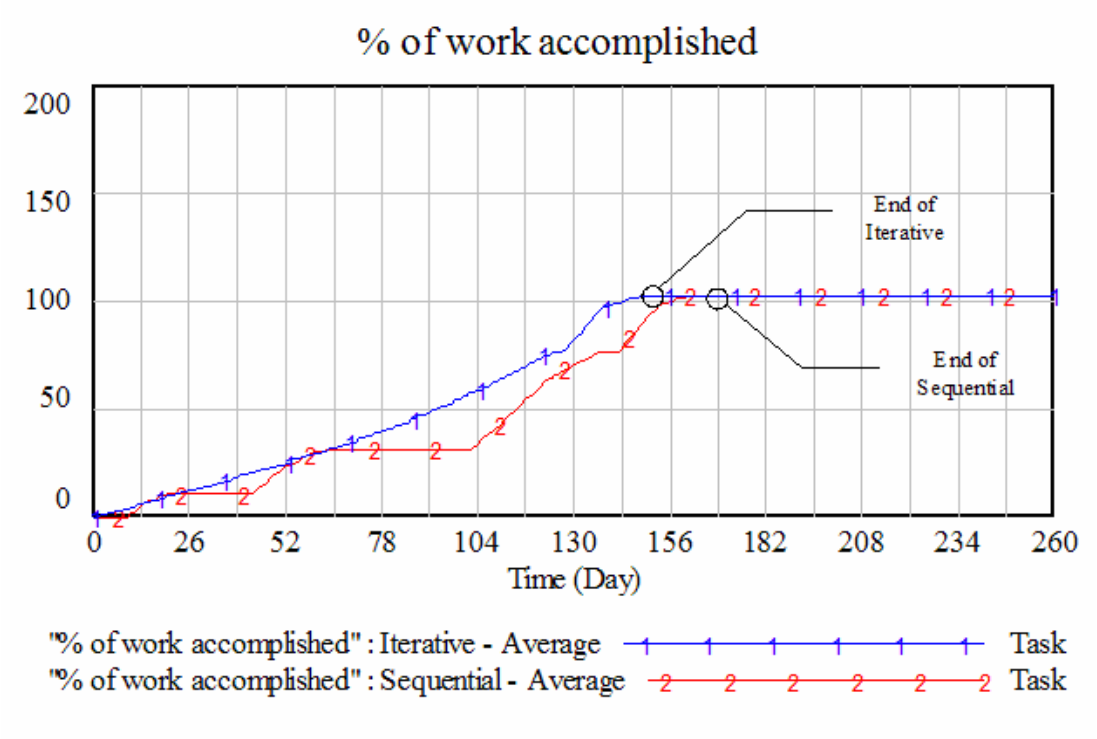

Figure 17.% of work accomplished.

The variable *Accumulated rework*, that measures the number of tasks that were reworked, shows that the sequential approach sent a larger number of tasks to be reworked than the iterative approach. A larger number of tasks will generate more development work and that might explain the sequential approach's delay.

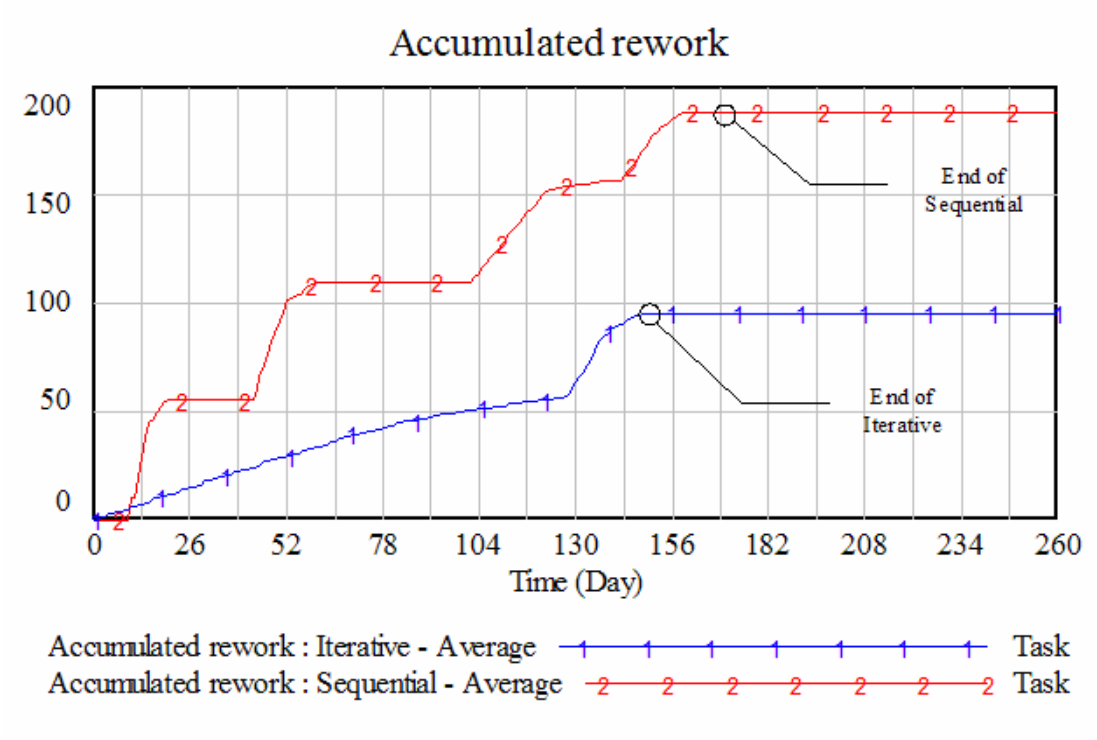

Figure 18. Accumulated rework.

However, to compare these numbers is somehow misleading because, as opposed to the iterative approach that merges analysis, design, and coding activities in a single phase, in the sequential approach the rework effort is different in each phase. Dividing the *Accumulated rework* by the *Phase size* allows us to make a fairer comparison between the values of the two approaches. Fig. 19 shows in fact that, when the size of each phase is taken into account, the iterative approach spends more effort in rework activities.

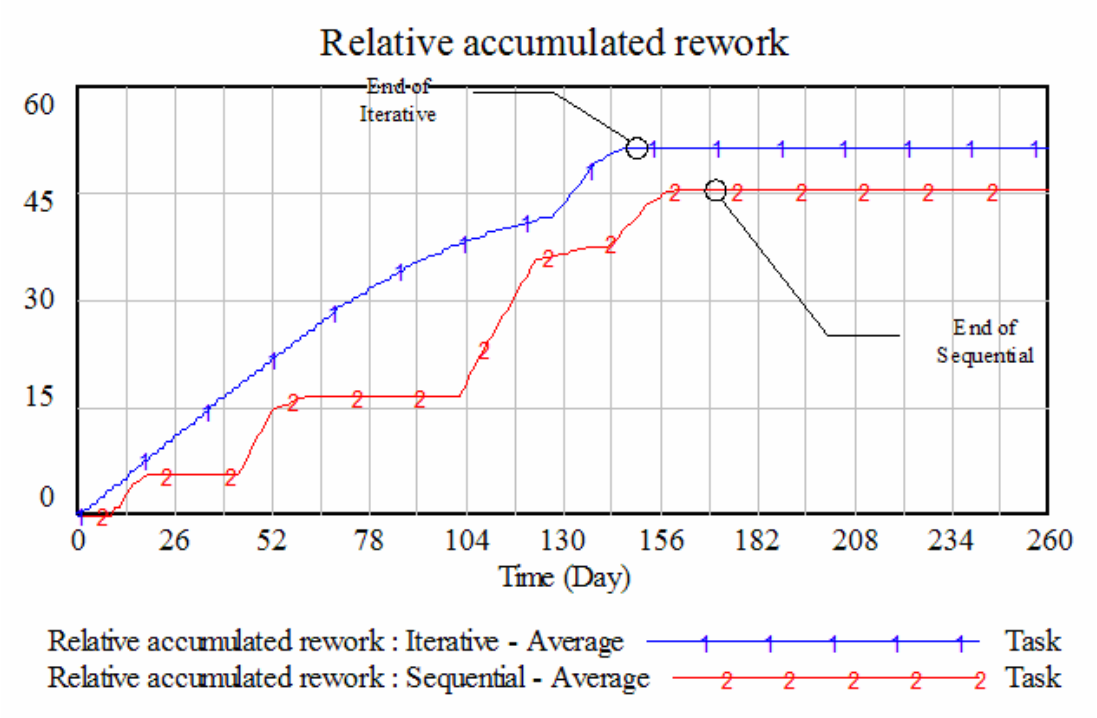

Figure 19. Relative accumulated rework.

Since both approaches in this model share a similar development rate, a larger number of reworked tasks in a smaller period of time suggests a better use of time. To confirm this hypothesis a new variable was created. This variable, *% of effective development working time*, shows what percentage of the total available time the development team had was spent doing development tasks and not waiting for new tasks nor supporting verification tasks.

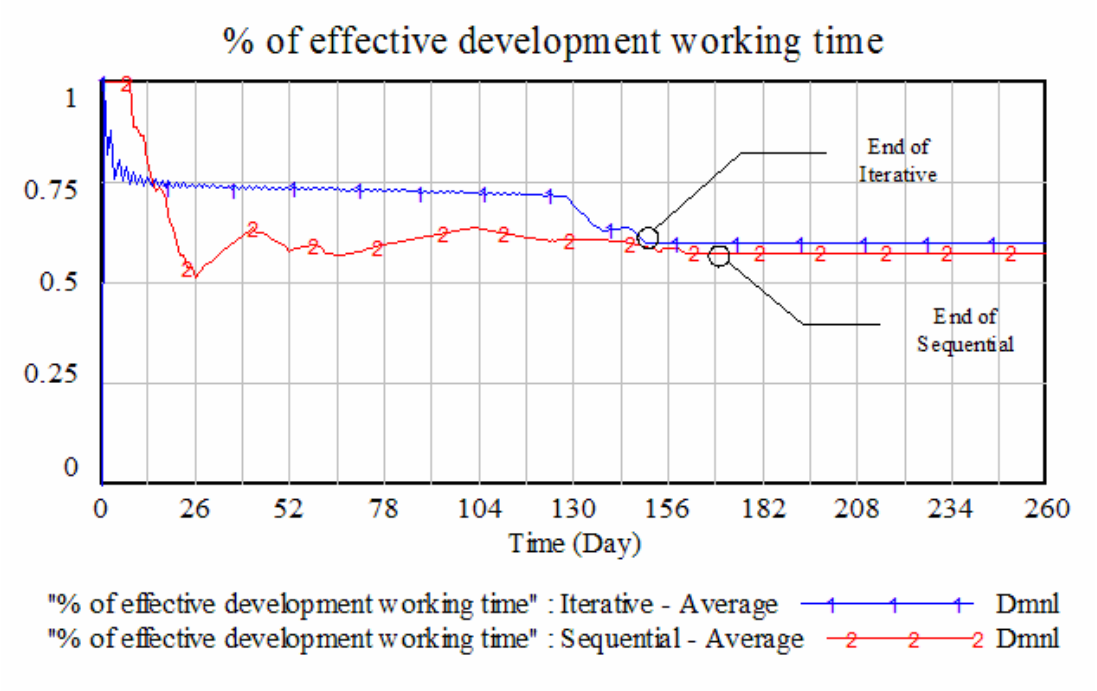

Figure 20. % of effective development working time.

Fig. 20 shows that the iterative approach, except during the first week, had a more effective use of time. The short cycles of development helped to keep the team busier, either working on new tasks or fixing previous work. As result, this approach was able to deliver more rework in a shorter period of time. On the other hand, tasks in the sequential approach were verified only towards the end of the phase which caused an uneven distribution of working time and periods were the team was not fully occupied.

This explains the reasons why the iterative approach ends first but why it had a slightly lower quality still remains unclear. Tasks, after being verified, have only three possible destinations: if correct they are sent to *Tasks that don't need rework*, if they have errors and these are found they are sent to *Tasks to be done*, and if they have errors but these are not detected they are sent to *Tasks with errors undetected*. In the sequential approach (see Fig. 21) it is observed that the number of tasks with errors undetected shows a slight decrease during the first three phases (55 tasks in the first phase, 45 in the second and 25 in the third). Likewise, the last phase shows a decrease but this time it is more significant (3.5 tasks with

errors at the end). The iterative approach, in turn, had 29.8 tasks with undetected errors at the end of the first phase.

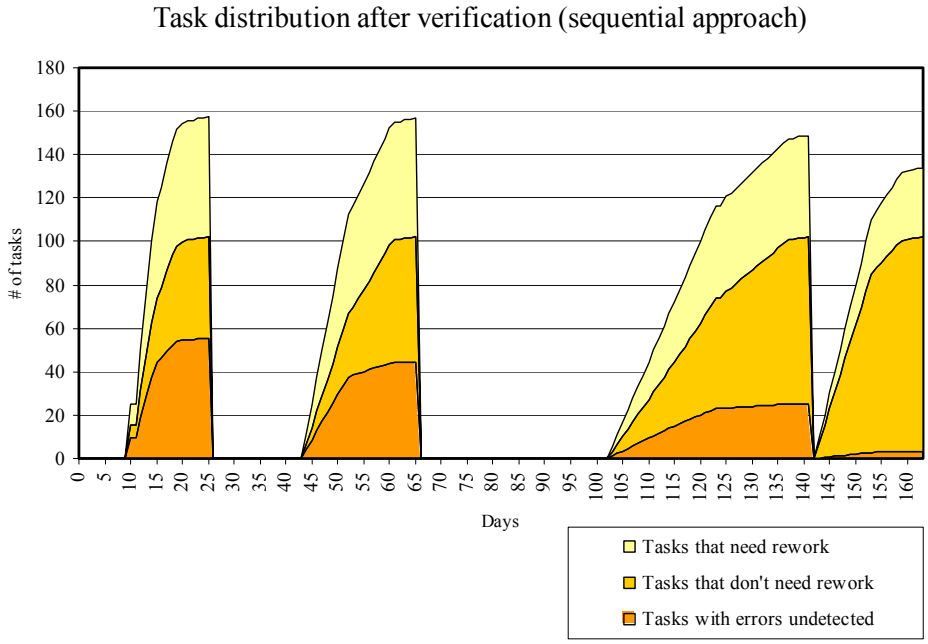

Figure 21. Task distribution after verification (sequential approach)

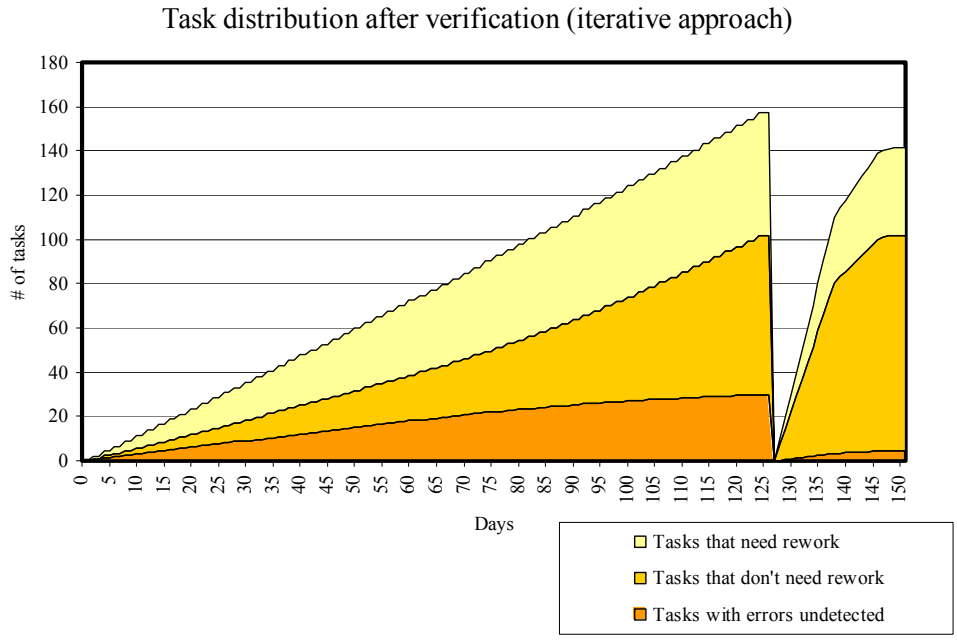

Figure 22. Task distribution after verification (iterative approach)

The % of tasks with errors undetected shows the evolution of the number of tasks that were approved even though they had errors. It only has values for those periods when tasks were being verified. Fig. 23 shows that the sequential approach had larger periods where only development tasks were performed. Towards the end of the coding phase both approaches had a similar % of tasks with errors undetected. However, due to the additional days needed to complete the project few additional tasks with errors were detected towards the end of the project in the sequential approach, which explains why in this simulation this approach had a final better quality.

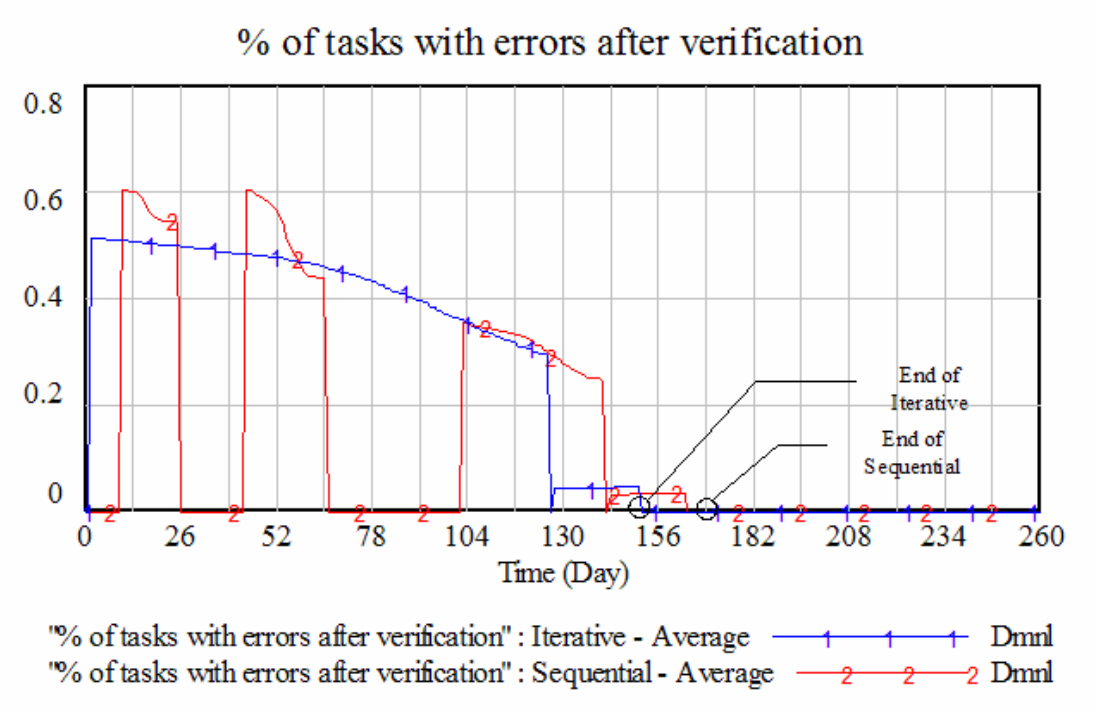

Figure 23. % of tasks with errors after verification.

Another claim of iterative methodologies' proponents is that these approaches help the team and the users to build a better understanding about the project faster. The data obtained from this simulation suggest that, at the beginning of the project, insight grows faster in the sequential mode. However, towards the middle of the project the insight developed in the iterative approach is already greater than that of the sequential approach. The initial slow rate of the iterative approach is probably associated with the lack of a previous analysis and design phases but the use of early code and verification helps to develop understanding in more steady fashion.

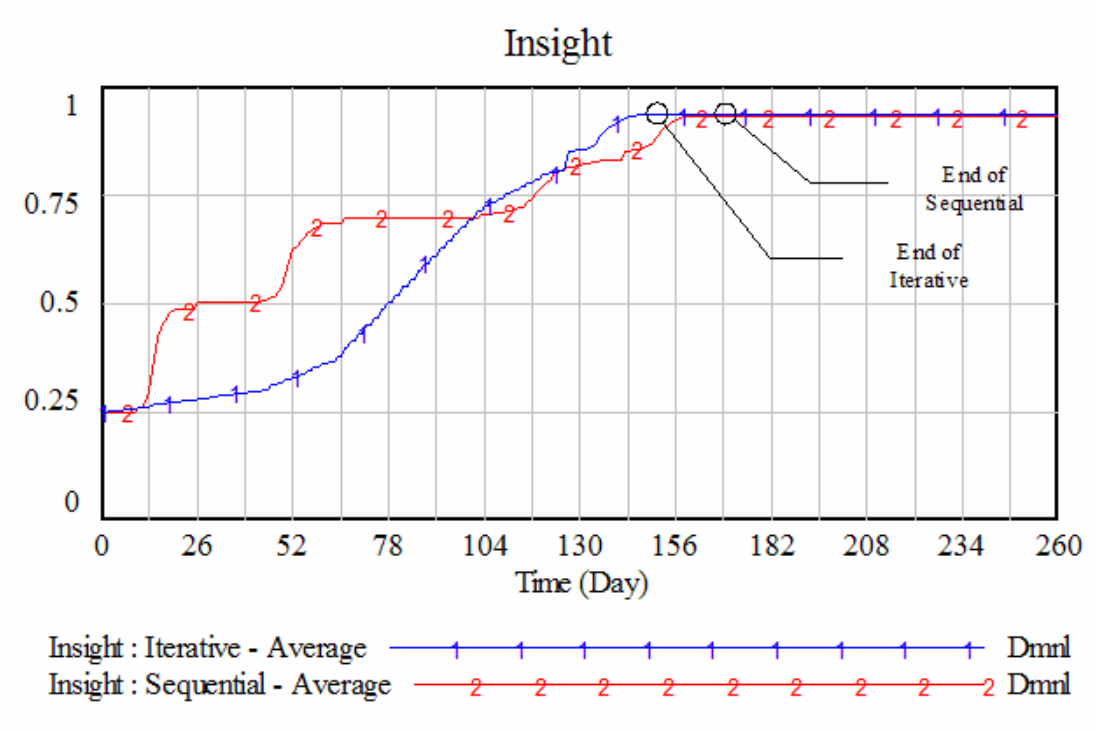

Figure 24. Insight.

V.1. Sensitivity Analysis

This section discusses the results obtained after performing a series of different scenarios using one-factor-at-a-time.

V.1.1. Changes in the Insight Development

To analyze the insight and its impact in the model it was assumed that during any phase of development the maximum level of insight could be reached after have been verified all the tasks twice and that more verification beyond that point don't contribute significantly to gain more insight. To analyze the sensitivity of this variable three different S-shape curves were proposed. The first, assumes that much insight can be gained since the beginning of the

project initial verification tasks. The second curve assumes that the insight development is equally distributed along the project. The third curve assumes that the insight grows slower at the beginning and builds up faster after all tasks have been verified at least one (see Fig. 25).

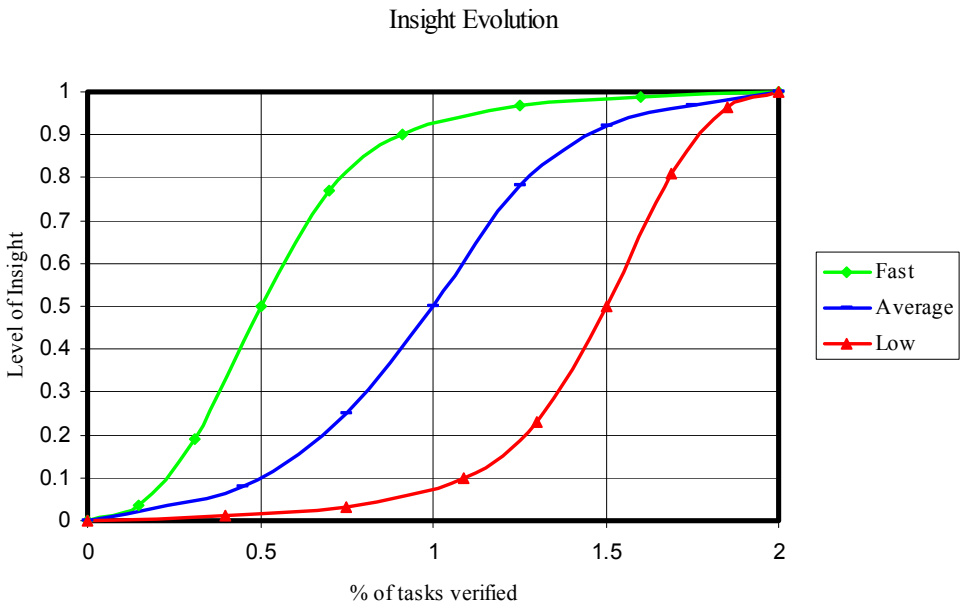

Figure 25. Insight evolution curves.

Fig. 26 and Table 6 show that a faster insight evolution helps to finish the project earlier and reduce the number of errors not detected at the end of the project. It is also observed that changes on the insight have a greater impact in the iterative approach than in the sequential approach.

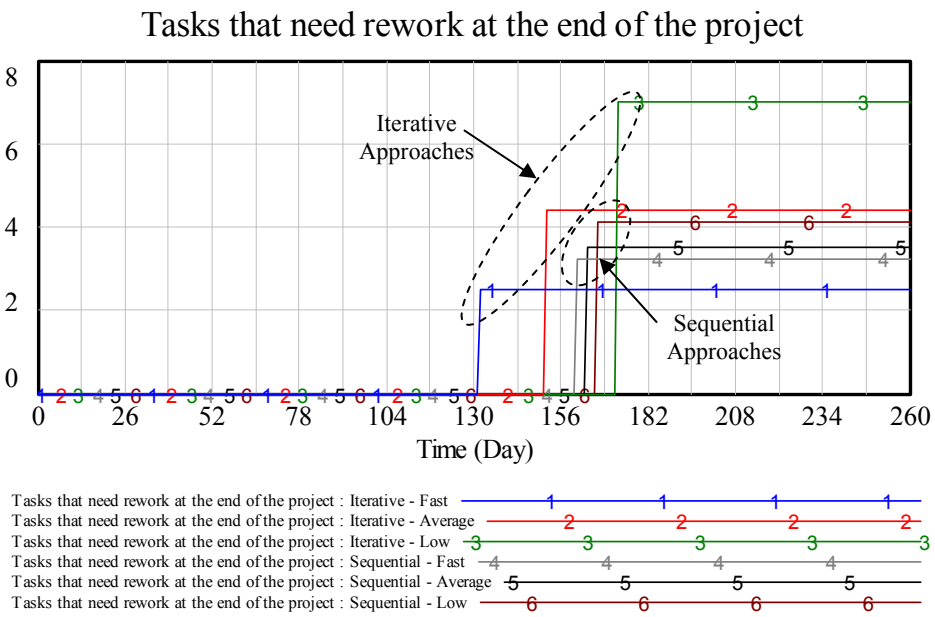

Figure 26. Insight evolution sensitivity.

	Number of days to completion	Tasks with error after completion
Iterative – Slow	173	7.0
Iterative – Average	152	4.4
Iterative – Fast	132	2.5
Average – Slow	167	4.1
Average – Average	164	3.5
Average – Fast	161	3.2

Table 6. Insight evolution sensitivity.

V.1.2. Changes in the Verification Period

One of the most relevant features of the iterative approach implemented in this model are the iterative cycles of development. According to new methodologies such as Extreme programming, very small cycles help the team to develop a faster understanding of the user's needs. Fig. 27 shows 5 different cycles of development: every day, every week, every two weeks, every three weeks and every month. It can be observed that, when other parameters

remain the same, larger verification periods cause delay in the project completion date and also increase the number of tasks with errors not detected at the end of the project. Even so, some of these cycles ended earlier than the sequential approach, which seems to support the claims made by methodologies such as extreme programming that short development cycles of integration and validation are more effective. Still the sequential approach produced apparently a better quality at the end.

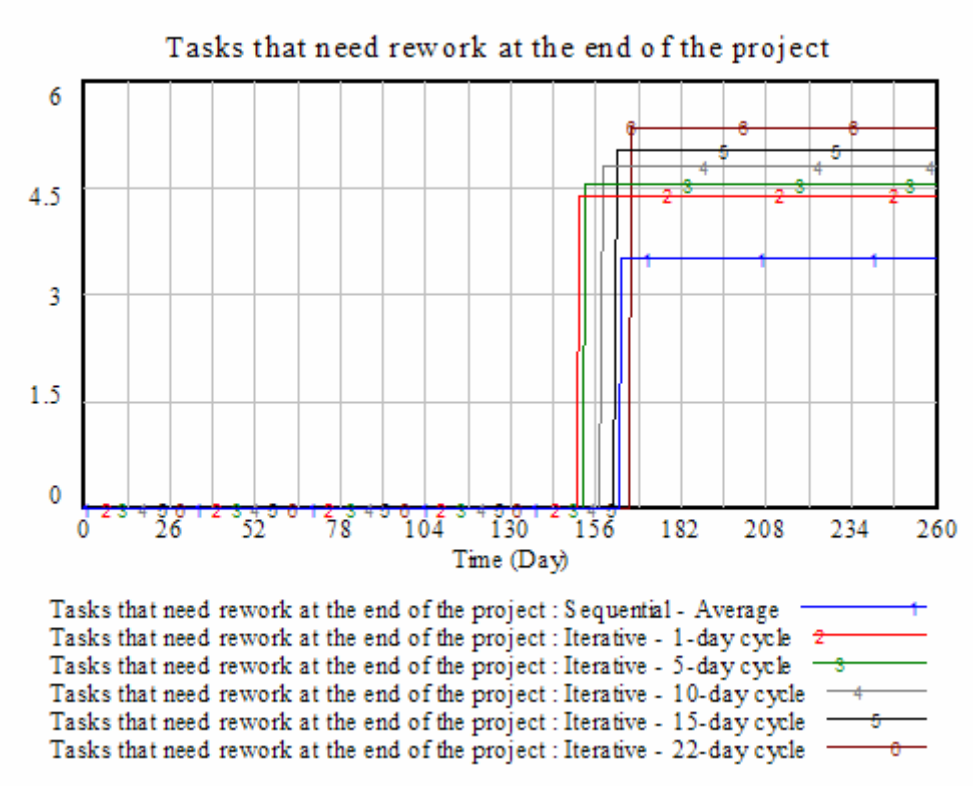

Figure 27. Verification period sensitivity.

	Number of days to completion	Tasks with error after completion
Sequential – Average	164	3.5
Iterative – 1–day cycle	151	4.4
Iterative – 5–day cycle	153	4.6
Iterative – 10–day cycle	158	4.8
Iterative – 15–day cycle	162	5.0
Iterative – 22–day cycle	167	5.3

Table 7. Verification period sensitivity.

V.1.3. Changes in the productivity

The model considers an increase in the productivity as result of the repetition of tasks. If the development team has a considerable expertise in the technology used to develop the project then the increase in the productivity is less significant. On the contrary, if the development team is not familiar with the type of project being developed, then there is more space for improvement and higher increases of productivity should be expected. To test the impact of productivity, they were tested three different values: 0%, 25%, and 50%.

Fig. 29 shows that higher increases in productivity reduced the development time in both cases. However, the impact of this parameter in the number of tasks with errors at the end of the project is not significant.

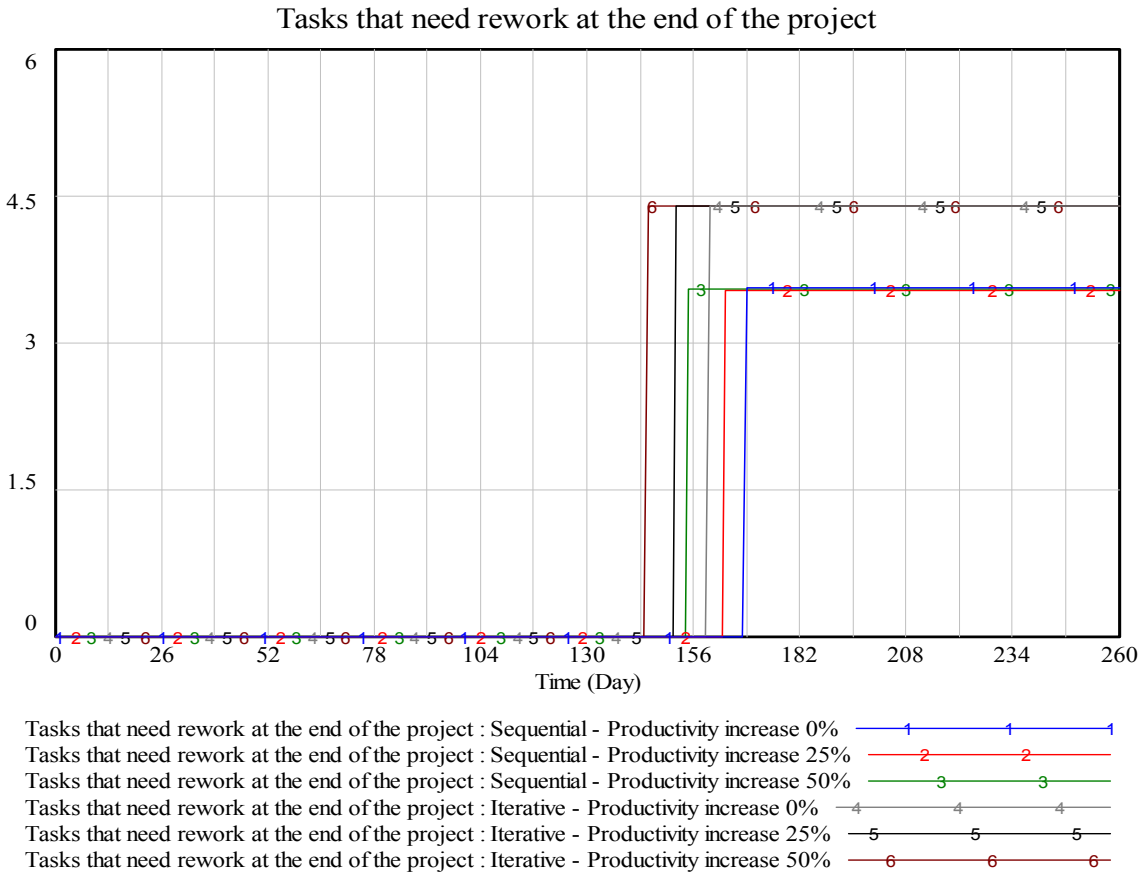

Figure 28. Productivity sensitivity.

	Number of days to completion	Tasks with error after completion
Sequential – Productivity 0%	169	3.5
Sequential – Productivity 25%	164	3.5
Sequential – Productivity 50%	155	3.5
Iterative – Productivity 0%	160	4.3
Iterative – Productivity 25%	152	4.3
Iterative – Productivity 50%	145	4.3

Table 8. Productivity sensitivity.

Chapter VI: Summary

The waterfall model was an important first step in the evolution of software management. It helps to highlight the need for a more carefully developed plan before a software project is initiated. Arguably, it has been the most influent approach in the short history of software. With some secondary variations, this model is still widely used in the industry.

The waterfall model, however, didn't address some important aspects of software development such as the need for flexibility and risk management. As an alternative to this model, the spiral model was suggested by the same people that made the waterfall model popular.

The spiral model focuses on risk management. It suggests having iterations in order to evaluate critical aspects of a project. Every cycle has a similar structure but with increasingly larger scope and duration. At the end of each cycle risk analysis is performed to decide weather the project should continue to a next iteration. This model works well in theory yet it was challenging to implement in practice. Critics say that, although a good model, it is also very complicated and should only be used by highly experienced managers.

The spiral model helped to promotes alternative styles to the waterfall model. Recent years have witnessed the emergence of new styles that claim more flexibility. Agile Methods and Extreme programming suggest not only a framework in terms of project planning and organization. They go a step further and also suggest practices to be applied in the daily work. These methodologies leverage the power of work team and fast feedback. However, as they themselves recognize, these methodologies are not to be applied in all cases.

Chapter VII: Conclusions and future work

VII.1. Conclusions

The first hypothesis of this work was that software development methodologies have a significant impact in success of software project. The literary research and the interviews support this affirmation. Likewise, the model shows how different approaches affect the quality and schedule of business application software. Therefore, this hypothesis is accepted. The second hypothesis was that most software methodologies can be broken down into two categories: sequential and iterative. Literary research showed that, although many formal methodologies might fall into these categories, there is also a number of hybrid models that featuring elements of both type of approaches and therefore cannot be categorized as either exclusive sequential or iterative. Therefore this hypothesis is rejected. The third hypothesis was that some features of these methodologies could be isolated and combined in order to study their impact on developing projects. Although the results of the sensitivity analysis suggest that this hypothesis is true, lack of real projects data using both approaches prevent us to confirm its validity. The fourth and last hypothesis was that some of these features can be combined in order to propose new approaches and improve management software project. The study of software evolution and, particularly, the emergence of hybrid models along with the results of the model developed strongly support this hypothesis, for why the last hypothesis might also be true.

Iterative and sequential software development methodologies have significantly different characteristics that make them more suitable for different type of projects. The understanding of these differences may facilitate the appropriate methodology selection for a particular project depending upon aspects such as the novelty, scope, flexibility, and quality. The iterative approach seems to offer a more effective use of time. As a result of their short iterations, idle time spent waiting for tasks to be reworked after verification activities is shorter and therefore a larger number of tasks can be worked with this approach. On the

contrary, in the sequential approach the separation of phases forces the team to wait until all the tasks have been developed and verified before a new phase can be started. Thus, the second half of each phase, when verification tasks begin, the idle time increases.

The iterative approach shows better results when insight can be developed fast since the beginning of the project. This scenario is typical of projects when the user is aware of his needs but is not familiar with the technology and, therefore, doesn't know with certainty what type of functionality the system may deliver. This is a common scenario for new technologies or COTS implementations. On the other hand, sequential approaches show more stable results in term of both time and quality. The sensitivity analysis shows that changes in insight development, verification period, or productivity have a smaller impact on the sequential approach than on the iterative approach. This suggests that when there is little certainty about the initial conditions of a project (in terms of team experience or insight) and there is not much flexibility regarding the delivery date and the final quality, the sequential is a less risky option.

In both approaches rework accounts for more than the half of the project duration. Early identification of rework is an excellent strategy to reduce the project's duration. Undetected errors not only delay the completion of the project but they also need more time to be fixed the later they are detected. The iterative approach shows that the short iterations are an effective mechanism to detect errors earlier and, therefore, to shorten the project.

VII.2. Future Work

The system dynamics model developed didn't include some elements that should improve the accuracy of the results. Some examples are: the impact of motivation on productivity, phase overlapping in the sequential approach, different types of complexity for tasks, willingness to add or modify tasks, and willingness to hire new employees. Benchmarking against a set of similar projects developed using both methodologies would also help to calibrate the model and increase the validity of the results.

Regarding the sensitivity analysis, additional work looking at other variables such as the relative size of phases, the experience of the developers, and the verification accuracy should be followed. Likewise, other design of experiments such as full factorial or orthogonal array should also be investigated in the future to understand better the impact that different scenarios have on software methodologies.

Some additional topics that might follow this work are: cost-benefit analysis of software development methodologies, study of hybrid models to increase productivity and quality, and the price of flexibility in software.

Bibliography

1. Aron J. D., “Estimating resources for Large Programming Systems”, Litton Educational Publishing, Inc., 1976
2. Beck Kent, “Extreme Programming explained”, Addison-Wesley, 2000
3. Beynon-Davies P., Holmes S., “Integrating rapid application development and participatory design”, IEE Proceedings Software, Vol. 145, No. 4, August 1998
4. Blanchard Benjamin S., Fabrycky Wolter J., “Systems Engineering and Analysis”, Third Edition, Prentice-Hall, 1998
5. Boehm Barry W., Bose Prasanta, “A Collaborative Spiral Software Process Model Based on Theory W”, USC Center for Software Engineering, University of Southern California, 1994.
6. Boehm Barry W., “Software Risk Management”, IEEE Computer Society Press, 1989
7. Brooks Frederick, “The Mythical Man-Month”, Addison Wesley Longman, 1995
8. Constantine Larry L., “Beyond Chaos: The expert Edge in Managing Software Development”, Addison-Wesley, 2001
9. Conte S. D., Dunsmore H. E., and Shen V.Y., “Software Engineering Metrics and Models”, Benjamin/Cummings, 1986
10. Gibbs W. Wayt, “Software’s Chronic Crisis”, Scientific American, September 1994
11. Highsmith Jim, Cockburn Alistair, “Agile Software Development: The people factor”, Software Management, November 2001
12. Highsmith Jim, Cockburn Alistair, “Agile Software: The Business of Innovation”, Software Management, September 2001
13. Jones Capers, “Applied software measurement: assuring productivity and quality”, McGraw-Hill 1991
14. Kruger Charles, “Software Reuse”, ACM Computing Surveys, Vol. 24, No. 2, June 1992
15. Leveson Nancy, “Safeware: System Safety and Computers”, Addison-Wesley, 1995

16. Madnick Stuart, Abdel-Hamid Tarek K., “The dynamics of software project scheduling: a system dynamic perspective”, Center for Information Systems Research, Alfred P. Sloan School of Management, 1982.
17. McConnell Steve, “Code complete: a practical handbook of software construction”, Microsoft Press, 1993
18. McConnell Steve, “Rapid Development”, Microsoft, 1996
19. Paulk Mark, Curtis Bill, Chrissis Mary Beth, Weber Charles, “The Capability Maturity Model”, Software Engineering Institute, Carnegie Mellon University, 1993
20. Rakos John J., “Software project management for small to medium sized projects”, Prentice-Hall, 1990.
21. Really John P., Carmel Erran, “Does RAD Live Up?”, IEEE Software, September 1995
22. “Manifesto for Agile Software Development” web site (http://agilemanifesto.org/)

Appendixes

Appendix A. Survey

This survey was conducted to 12 software project managers from 5 different companies in Peru:

- Company A is a Bank with branches in several countries in South America.
- Company B is an international Non-profit organization that helps people through microlending.
- Company C is a Bank specialized in Retail Banking.
- Company D is a software company specialized in developing Business Applications for the Banking Industry
- Company E is a consulting company specialized in developing Business Intelligence projects.

1. In the last 2 years, in how many business applications software development have you participated?

Total	Average	Standard Dev
66	4.9	2.3

2. In average these projects …

Average	Standard Dev	Units
4.6	1.7	months
6.6	2.1	months
21.0	13.8	%
1.3	0.5	months
7.7	3.2	people
4.1	1.5	
1.8	1.2	months

3. How would you define a successful project?

A successful project is one that …	%
… satisfies the users' requirements	92%
… doesn't need budget extensions	58%

… finishes on time	50%
… exceeds user's expectations	25%
… creates value to the company	25%
… ends reasonably on time (less than 10% delay)	17%
… has a long useful life	17%
… has an excellent technical design	8%

4. According to your definition, what percentage of projects you were involved was successful?

Average	Standard Dev
56.2	29.9

5. What would you identify as the most important factor for this success?

Factor	%
Good specifications	42%
Project management	33%
Communication	25%
Technical Experience	8%
Planning	8%

6. According to your experience, what are the most common sources of problems in the development of business application software?

Problem	%
User specifications' ambiguity	89%
Unrealistic planning	80%
Changes in the specifications during planning	77%
Lack of technical experience	58%
Lack of a development methodology	58%
Poor risk management	52%
Parallel projects affecting team's productivity	48%
Low budgets	36%
Poor coordination to allocate resources	34%
Poor testing	28%
Lack of motivation	25%
Other technical problems	21%

7. According to your experience, how often do software projects suffer delays?

Frequency	%
Never	0%
0 - 25%	0%
25% - 50%	0%
50% - 75%	0%
Always	100%

8. What are the causes of this delay?

Causes	%
Poor estimation	67%
Requirements changes	50%
Poor quality of deliverables during analysis and design	8%
Poor management of user expectations	8%
Lack of communication between users and development team	8%

9. What would you recommend to address this issue?

Causes	%
Improve estimation	50%
Improve tracking	25%
Train users	17%
Train developers	17%

10. Which phase would you characterize as the most critical?

Phase	%
Analysis and Design	50%
Design	20%
Testing	20%
All	10%

11. Which phase is the most likely to experience delays?

Phase	%
Coding	75%
Analysis	25%

12. Do you use a formal development methodology?

Answer	%
No	33%
Yes	67%

13. If your previous answer was yes, please mention which methodology do you use?

Methodology	%
Capability Maturity Model (CMM)	8%
Microsoft Solutions Framework (MSF)	17%
Program Management Office (PMO)	42%

14. Do you use sequential or iterative methodologies?

Methodology	%
Sequential	83%
Iterative	0%
Hybrid	17%

15. Please rank the following problems according to their impact in the project

Problem	Rank	%
Ambiguity in specifications	1	91%
Lack of a more realistic estimation	2	76%
Adding functionality during coding	3	76%
Lack of technical experience	4	59%
Lack of a formal development methodology	5	52%
Poor risk management	6	47%
Multiple projects in parallel	7	45%
Lack of funding	8	38%
Lack of coordination between users and the development team	9	37%
Poor testing	10	31%
Lack of motivation	11	28%
Unexpected technical problems	12	27%

16. Please rank problems according to their frequency

	Rank	%
Ambiguity in specifications	1	91%
Adding functionality during coding	2	89%
Lack of a more realistic estimation	3	76%
Poor risk management	4	52%
Lack of technical experience	5	51%
Lack of a formal development methodology	6	47%
Poor testing	7	42%
Lack of funding	8	38%
Lack of motivation	9	38%
Unexpected technical problems	10	37%
Multiple projects in parallel	11	36%
Lack of coordination between users and the development team	12	35%

17. State your positions regarding the following statements (-2=Strongly Disagree, 2=Strongly Agree)

Statement	Average
In general, business applications software development require extensions of budget or schedule	1.7
In general, development issues could have been avoided with a better management practice	1.4
In general, after they are put in production, users identify many problems that should have been captured during testing	1.3
Due to pressure to end the project on time, some tasks such as documentation are usually deprioritized	1.3
To estimate the ROI of a software application is usually an ambiguous task	0.9
During Coding, it is common to add functionality that wasn't	0.9

specified during analysis	
To add new people to a late project doesn't help to end a project on time	0.3
It's better to postpone the end of a project than to postpone some tasks to a new phase	0.1
In general, it's hard for users and analysts to identify and translate into functional specifications all the users' needs	0.1

Appendix B. System Dynamics Model

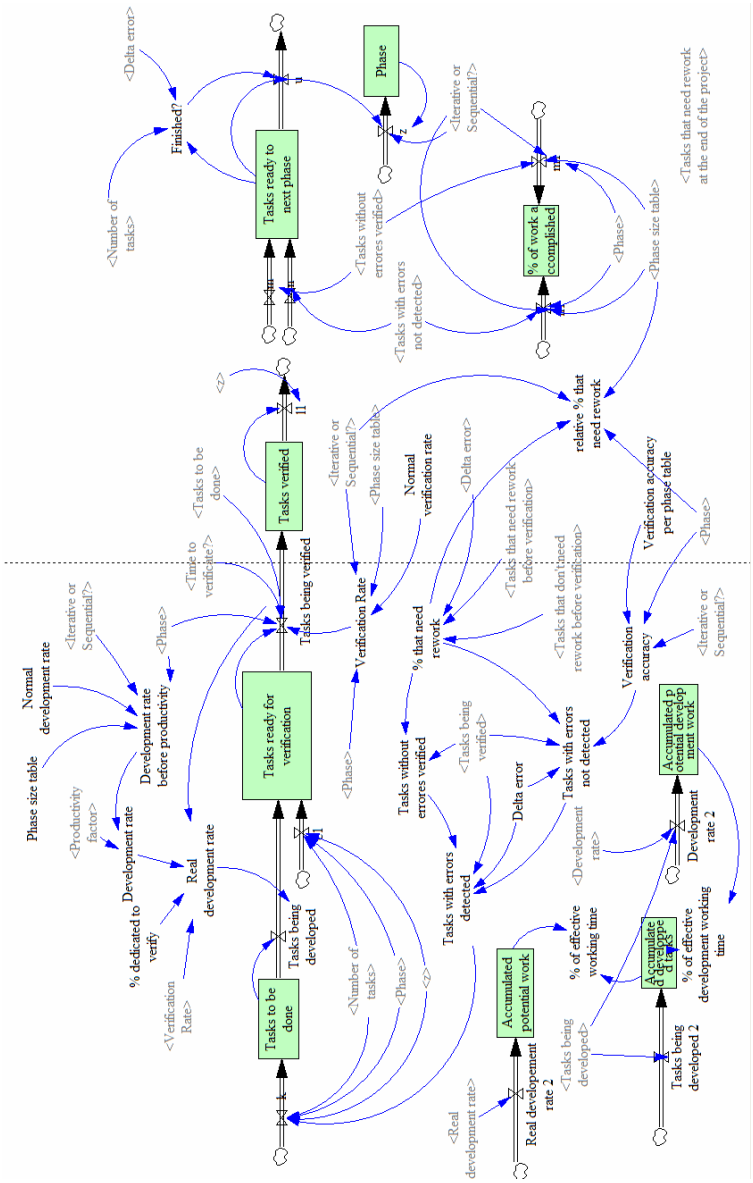

Figure 29. System Dynamics Model – Part I

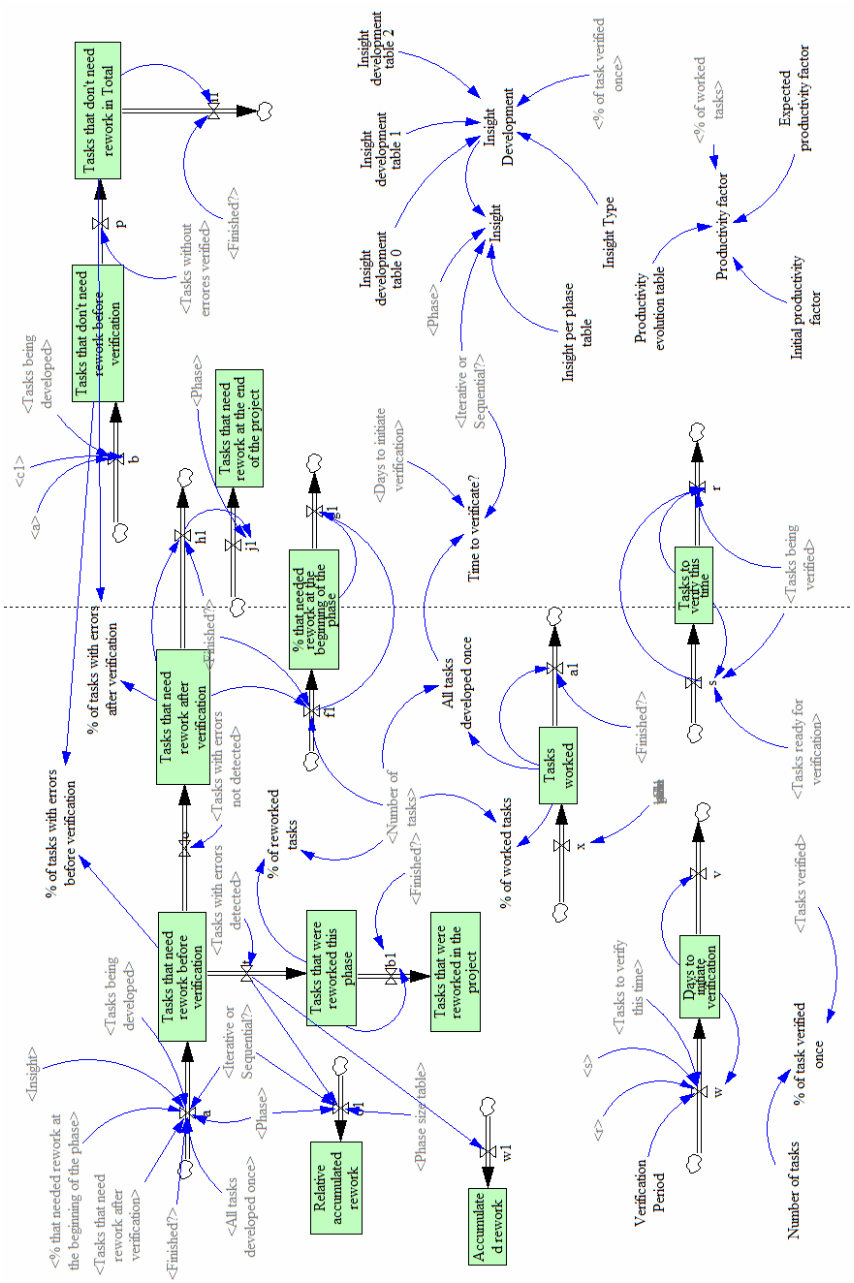

Figure 30. System Dynamics Model – Part II

Appendix C. Stocks, Flows and Variables of the Model

Variable	Definition	Dimension
% dedicated to verify	=0.75	Dmnl
% of effective development working time	=if then else(Accumulated potential development work>0,Accumulated developed tasks/Accumulated potential development work,0)	Dmnl
% of effective working time	=if then else (Accumulated potential work>0,Accumulated developed tasks/Accumulated potential work,0)	Dmnl
% of reworked tasks	=Tasks that were reworked this phase/Number of tasks	Dmnl
% of task verified once	=Tasks verified/Number of tasks	Dmnl
% of tasks with errors after verification	=if then else(Tasks that don't need rework in Total>0:OR:Tasks that need rework after verification>0,Tasks that need rework after verification/(Tasks that don't need rework in Total+Tasks that need rework after verification),0)	Dmnl
% of tasks with errors before verification	=if then else(Tasks that don't need rework before verification>0:AND:Tasks that need rework before verification>0,Tasks that need rework before verification/(Tasks that don't need rework before verification+Tasks that need rework before verification),0)	Dmnl
% of work accomplished	= INTEG (m1+n1,0)	Task
% of worked tasks	=if then else (Number of tasks>0,Tasks worked/Number of tasks,0)	Dmnl
% that need rework	=if then else (Tasks that don't need rework before verification>Delta error :AND: Tasks that need rework before verification>Delta error,Tasks that need rework before verification/(Tasks that need rework before verification+Tasks that don't need rework before verification),0)	Dmnl
% that needed rework at the beginning of the phase	= INTEG (f1-g1,0)	Dmnl
a	=if then else ("Finished?"=0, if then else (All tasks developed once=0,Tasks being developed*"% that needed rework at the beginning of the phase"+Tasks being developed*(1-"% that needed rework at the beginning of the phase")*(1-Insight),Tasks being developed*(1-Insight)), if then else (("Iterative or Sequential?"=0:AND:Phase=2) :OR: ("Iterative or Sequential?"=1:AND:Phase=0),Tasks that need rework after verification,Tasks being developed*"% that needed rework at the beginning of the phase"+Tasks being developed*(1-"% that needed rework at the beginning of the phase")*(1-Insight)))	Task/Day
a1	=if then else ("Finished?"=1,Tasks worked,0)	Dmnl
Accumulated developed	= INTEG (Tasks being developed 2,0)	Task

tasks		
Accumulated potential development work	= INTEG (Development rate 2,0)	Task
Accumulated potential work	= INTEG (Real developement rate 2,0)	Task
Accumulated rework	= INTEG (w1,0)	Task
All tasks developed once	=if then else (Number of tasks>0,if then else(Tasks worked/Number of tasks>=1,1,0),0)	Dmnl
b	=(Tasks being developed+c1)-a	Task/Day
b1	=if then else ("Finished?"=1,Tasks that were reworked this phase,0)	Task/Day
c1	=if then else (z>0:AND:z+Phase=3,Number of tasks,0)	Task/Day
Days to initiate verification	= INTEG (+w-v,Verification Period)	Dmnl
Delta error	=0.1	Dmnl
Development rate	=Development rate before productivity*Productivity factor	Dmnl
Development rate 2	=if then else (Tasks being developed>0,Development rate,0)	Task/Day
Development rate before productivity	=if then else("Iterative or Sequential?"=1:AND:Phase<3, Normal development rate/(1-Phase size table(3)),Normal development rate/Phase size table(Phase))	Dmnl
Expected productivity factor	=1.25	Dmnl
f1	=if then else ("Finished?"=1,Tasks that need rework after verification/Number of tasks,0)	Dmnl
Finished?	=if then else (Tasks ready to next phase+Delta error>=Number of tasks,1,0)	Dmnl
g1	=if then else (f1>0,"% that needed rework at the beginning of the phase",0)	Dmnl
h1	=if then else ("Finished?"=1,Tasks that need rework after verification,0)	Task/Day
i1	=if then else ("Finished?"=1,Tasks that don't need rework in Total,0)	Task/Day
Initial productivity factor	=1	Dmnl
Insight	=if then else ("Iterative or Sequential?"=1:AND:Phase<3,Insight per phase table(0)+(Insight Development)*(Insight per phase table(3)-Insight per phase table(0)),Insight per phase table(Phase)+(Insight Development)*(Insight per phase table(Phase+1)-Insight per phase table(Phase)))	Dmnl
Insight Development	=if then else (Insight Type=0,Insight development table 0("% of task verified once"),if then else(Insight Type=1,Insight development table 1("% of task verified once"),Insight development table 2("% of task verified once")))	Dmnl
Insight development table 0	=([(0,0)-(2,1)],(0,0),(0.4,0.01),(0.75,0.03),(1.09,0.1),(1.3,0.23),(	Dmnl

	1.5,0.5),(1.69,0.81),(1.85,0.965),(2,1))	
Accumulated developed tasks	= INTEG (Tasks being developed 2,0)	Task
Accumulated potential development work	= INTEG (Development rate 2,0)	Task
Accumulated potential work	= INTEG (Real developement rate 2,0)	Task
Accumulated rework	= INTEG (w1,0)	Task
All tasks developed once	=if then else (Number of tasks>0,if then else(Tasks worked/Number of tasks>=1,1,0),0)	Dmnl
b	=(Tasks being developed+c1)-a	Task/Day
b1	=if then else ("Finished?"=1,Tasks that were reworked this phase,0)	Task/Day
c1	=if then else (z>0:AND:z+Phase=3,Number of tasks,0)	Task/Day
Insight development table 1	=([(0,0)-(2,1)],(0,0),(0.5,0.08),(0.75,0.2),(1,0.5),(1.25,0.78),(1.5,0.92),(1.75,0.97),(2,1))	Dmnl
Insight development table 2	=([(0,0)-(2,1)],(0,0),(0.15,0.035),(0.31,0.19),(0.5,0.5),(0.7,0.77),(0.91,0.9),(1.25,0.97),(1.6,0.99),(2,1))	Dmnl
Insight per phase table	=([(0,0)-(4,1)],(0,0.25),(0.990826,0.5),(2,0.7),(3,0.85),(4,0.95))	Dmnl
Insight Type	=1	Dmnl
Iterative or Sequential?	=1	Dmnl
j1	=if then else(Phase=3:AND:h1>0,h1,0)	Task/Day
k	=if then else (z>0:AND:(Phase+z)<3,Number of tasks,Tasks with errors detected)	Task/Day
l1	=if then else(z>0,Tasks verified,0)	Task/Day
m	=Tasks without errors verified	Task/Day
m1	=if then else ("Iterative or Sequential?"=1:AND:Phase=0,Tasks without errors verified*(1-Phase size table(3)),Tasks without errors verified * Phase size table(Phase))	Task/Day
N	=Tasks with errors not detected	Task/Day
n1	=if then else ("Iterative or Sequential?"=1:AND:Phase=0,Tasks with errors not detected*(1-Phase size table(3)),Tasks with errors not detected * Phase size table(Phase))	Task/Day
Normal development rate	=1.23	Dmnl
Normal verification rate	=2.5	Dmnl
Number of tasks	=102	Task
o	=Tasks with errors not detected	Task/Day
o1=if then else ("Iterative or Sequential?"	=1:AND:Phase=0,t*(1-Phase size table(3)),t*Phase size table(Phase))	Task/Day
p	=Tasks without errors verified	Task/Day
Phase	= INTEG (z,0)	Dmnl
Phase size table	=([(0,0)-(3,1)],(0,0.1),(1,0.2),(2,0.45),(3,0.25))	Dmnl

Productivity evolution table	=([(0,0)-(2,1)],(0,0),(0.25,0.04),(0.5,0.1),(0.75,0.25),(1,0.5),(1.25,0.75),(1.5,0.9),(1.75,0.96),(2,1))	Dmnl
Productivity factor	=Initial productivity factor+Productivity evolution table("% of worked tasks") *(Expected productivity factor-Initial productivity factor)	Dmnl
r	=if then else (Tasks to verify this time+s>=Tasks being verified,Tasks being verified,Tasks to verify this time)	Task/Day
Real developement rate 2	=Real development rate	Task/Day
Real development rate	=if then else(Tasks being verified>0,Development rate*(1-"% dedicated to verify"*(Tasks being verified/Verification Rate)),Development rate)	Dmnl
relative % that need rework	=if then else ("Iterative or Sequential?"=1:AND:Phase=0,"% that need rework"*(1-Phase size table(3)),"% that need rework"*Phase size table(Phase))	Dmnl
Insight development table 1	=([(0,0)-(2,1)],(0,0),(0.5,0.08),(0.75,0.2),(1,0.5),(1.25,0.78),(1.5,0.92),(1.75,0.97),(2,1))	Dmnl
Insight development table 2	=([(0,0)-(2,1)],(0,0),(0.15,0.035),(0.31,0.19),(0.5,0.5),(0.7,0.77),(0.91,0.9),(1.25,0.97),(1.6,0.99),(2,1))	Dmnl
Insight per phase table	=([(0,0)-(4,1)],(0,0.25),(0.990826,0.5),(2,0.7),(3,0.85),(4,0.95))	Dmnl
Insight Type	=1	Dmnl
Iterative or Sequential?	=1	Dmnl
j1	=if then else(Phase=3:AND:h1>0,h1,0)	Task/Day
k	=if then else (z>0:AND:(Phase+z)<3,Number of tasks,Tasks with errors detected)	Task/Day
l1	=if then else(z>0,Tasks verified,0)	Task/Day
m	=Tasks without errors verified	Task/Day
m1	=if then else ("Iterative or Sequential?"=1:AND:Phase=0,Tasks without errors verified*(1-Phase size table(3)),Tasks without errors verified * Phase size table(Phase))	Task/Day
N	=Tasks with errors not detected	Task/Day
n1	=if then else ("Iterative or Sequential?"=1:AND:Phase=0,Tasks with errors not detected*(1-Phase size table(3)),Tasks with errors not detected * Phase size table(Phase))	Task/Day
Normal development rate	=1.23	Dmnl
Normal verification rate	=2.5	Dmnl
Number of tasks	=102	Task
o	=Tasks with errors not detected	Task/Day
o1=if then else ("Iterative or Sequential?"	=1:AND:Phase=0,t*(1-Phase size table(3)),t*Phase size table(Phase))	Task/Day

p	=Tasks without errors verified	Task/Day
Relative accumulated rework	= INTEG (o1,0)	Task
S	=if then else (Tasks being verified>0:AND:Tasks to verify this time=0,Tasks ready for verification,0)	Task/Day
T	=Tasks with errors detected	Task/Day
Tasks being developed	=if then else (Tasks to be done>Real development rate,Real development rate,if then else (Tasks to be done>0,Tasks to be done,0))	Task/Day
Tasks being developed 2	=Tasks being developed	Task/Day
Tasks being verified	=if then else ((Tasks ready for verification>1:AND:"Time to verificate?"=1):OR:Phase=3:OR:(Tasks to be done=0:AND:Tasks ready for verification>0),if then else (Tasks ready for verification>=Verification Rate,Verification Rate,Tasks ready for verification),0)	Task/Day
Tasks ready for verification	= INTEG (Tasks being developed-Tasks being verified+c1,0)	Task
Tasks ready to next phase	= INTEG (m+n-u,0)	Task
Tasks that don't need rework before verification	= INTEG (b-p,0)	Task
Tasks that don't need rework in Total	= INTEG (p-i1,0)	Task
Tasks that need rework after verification	= INTEG (o-h1,0)	Task
Tasks that need rework at the end of the project	= INTEG (j1,0)	
Tasks that need rework before verification	= INTEG (a-o-t,0)	Task
Tasks that were reworked in the project	= INTEG (b1,0)	Task
Tasks that were reworked this phase	= INTEG (t-b1,0)	Task
Tasks to be done	= INTEG (k-Tasks being developed,Number of tasks)	Task
Tasks to verify this time	= INTEG (s-r,0)	Task
Tasks verified	= INTEG (Tasks being verified-l1,0)	Task
Tasks with errors detected	=if then else (Tasks being verified>Delta error, Tasks being verified-Tasks without errors verified-Tasks with errors not detected,0)	Dmnl
Tasks with errors not detected	=if then else (Tasks being verified>Delta error,Tasks being verified*"% that need rework"*(1-Verification accuracy),Tasks being verified*"% that need rework")	Dmnl
Tasks without errors verified	=Tasks being verified*(1-"% that need rework")	Dmnl
Tasks worked	= INTEG (x-a1,0)	Task/Day
Time to verificate?	=if then else ("Iterative or Sequential?"=1,if then else (Days to initiate verification=0,1,0),if then else(All tasks developed once=1,1,0))	Dmnl

u	=if then else("Finished?"=1, Tasks ready to next phase , 0)	Task/Day
v	=if then else (Days to initiate verification>0,1,0)	Dmnl
Verification accuracy	=if then else ("Iterative or Sequential?"=1:AND:Phase<>3,Verification accuracy per phase table(2),Verification accuracy per phase table(Phase))	Dmnl
Verification accuracy per phase table	=([(0,0)-(3,1)],(0,0.5),(1,0.55),(2,0.65),(3,0.9))	Dmnl
Verification Period	=1	Dmnl
Verification Rate	=if then else ("Iterative or Sequential?"=1,if then else(Phase=3,Normal verification rate/Phase size table(Phase),Normal verification rate/(1-Phase size table(3))),Normal verification rate/Phase size table(Phase))	Dmnl
w	=if then else (((Tasks to verify this time=0):OR:(Tasks to verify this time=r)):AND:Days to initiate verification=0:AND:((s=0):OR:(r=s:AND:Tasks to verify this time=0)),Verification Period,0)	Dmnl
w1	=t	Task
x	=Tasks being developed	Task/Day
z	=if then else(u>0:AND:Phase<3, if then else ("Iterative or Sequential?"=1,3,1) , 0)	Dmnl

Table 9. List of variables of the Model

Appendix D. Project Example

The following tables show the calculation of the number of function points for small project of medium complexity. These values are calculated identifying the number and type of components the project has (see Table10). Then these values are multiplied by the factor assigned to each type of component (see Table 11). The sum of the results defined the number of function-points (see Table 12).

Complexity	Inputs	Outputs	Inquiries	Logical internal files	External interface files
Low	2	1	2	1	1
Medium	2	2	2	1	1
High	2	1	2	0	0
Subtotal	6	4	6	2	2
Total					40

Table 10. Components of the example project

Complexity	Inputs	Outputs	Inquiries	Logical internal files	External interface files
Low	3	4	3	7	5
Medium	4	5	4	10	7
High	6	7	6	15	10

Table 11. Conversion factor to calculate adjusted function-point

Function Points	Inputs	Outputs	Inquiries	Logical internal files	External interface files
Low	6	4	6	7	5
Medium	8	10	8	10	7
High	12	7	12	0	0
Unadjusted function-point					102
Influence multiplier					1
Adjusted function-point total					102

Table 12. Adjusted function-point

The suggested duration of the project is calculated using the quality level exponent. An average business organization will develop this project in approximately 7.3 months (see Table 13).

Best in class	Average	Worst in class
0.41	0.43	0.46
$102^{0.41}$=6.7	$102^{0.43}$=7.3	$102^{0.46}$=8.4

Table 13. Project duration estimation

Appendix E. A closer look at Validation Phase in Company A

This section presents an additional system dynamics model that focuses on the last phase of a project. This model doesn't differentiate between sequential and iterative approaches but it helps to understand what the driven forces are behind this phase and how they can be adjusted to control it better.

The model has three parts. The purpose of the first part (see Fig. 31) is to model how tasks get transferred from the developers to the testers. Upon error corrections these rework tasks will be returned to the stock (tasks to be checked) for another testing cycle.

The number of cycles repeated will be captured by the stock (testing cycles) through the new cycle rate which is a function of task submission, every time the tasks are passed from the stock (tasks checked) to the stock (tasks to be checked) it goes through the task submission rate which serves as a counter for the stock (testing cycles).

The auxiliary variable (test time per cycle) is a function of the table (testing speed) and the stock (testing cycles) that controls the checking rate.

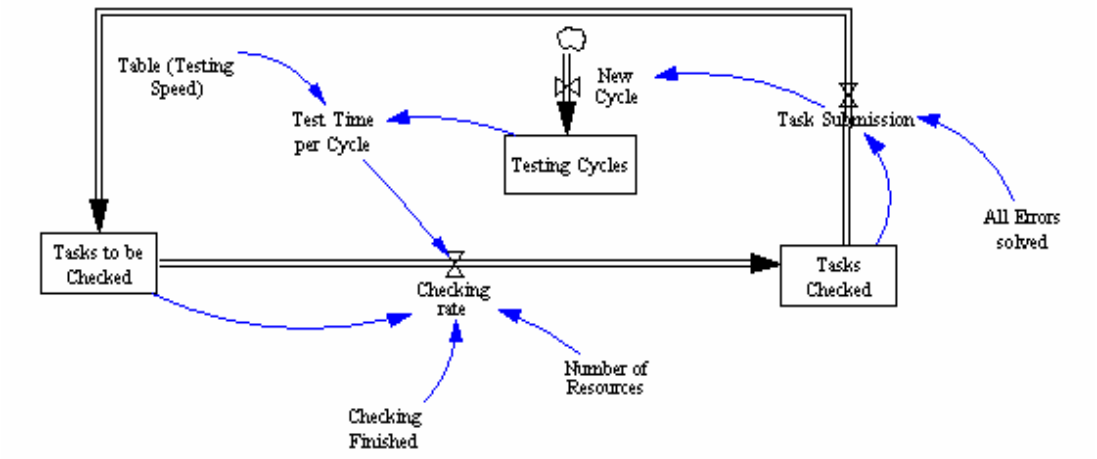

Figure 31. Testing cycle

The second part of the model focuses on the rework cycle, from identifying the coding errors to solving the errors (see Fig 32). Coding error rate, a function of error introduction rate and error submission, defines how many errors are generated. These errors are stored as undiscovered errors first. Then they become discovered errors via the error discovery rate that is a function of the table (probability of detection), error density, checking rate, checking finished, undiscovered errors to the stock (discovered errors). It then passes to the stock (error solved) via the error rework rate which is a function of the number of development resources, rework rate, and discovered errors. From the stock (error solved) it goes to the

stock (error inventory) via the error submission rate which is a function of the errors solved and all errors were solved.

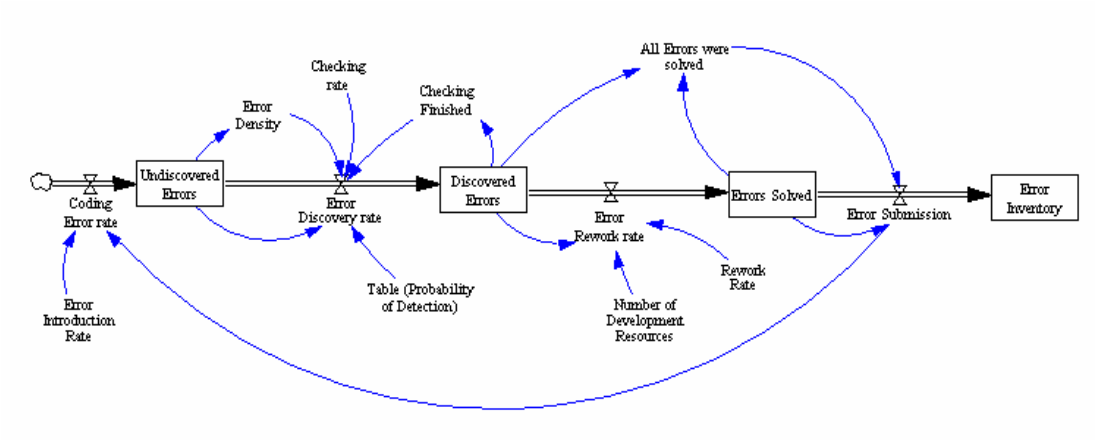

Figure 32. Rework Cycle

The third part of the model captures the total time spent on the phase (see Fig 33). The Stock and Flow diagram starts with a source via the time checking rate which is a function of checking rate and time step to the stock (cycle time). It then passes to the stock (total cycle time) via the checking end rate which is a function of the cycle time and checking finished.

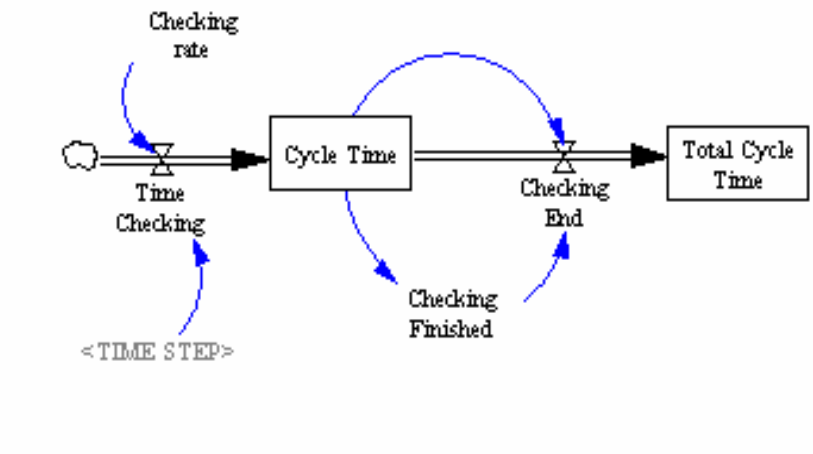

Figure 33. Time Control

Using information provided by company A (see Appendix A) a total of 14 simulations were performed to analyze how changing their current parameters cost and testing time improved. Thus, the model has four main parameters that allow simulating different scenarios variables. These are the: the number of testers, the number of developers, the maximum number of errors per cycle, the maximum number of days per cycle. To control the duration of testing cycles this model assumes the following policy: cycles finish whenever the maximum

number of errors or the maximum number of days are reached. When one of those events happens tasks testing is stopped and waits again until development is ready.

All simulations considered the average scenario of a project in Company A which includes: 2.5 Testers, 6 developers, and 100 tasks.

The metrics used to compare the results of each simulation were:

- Total time spent on work: defined as the total days the developers and the testers were working.
- Work Days: defined as the total number of days required to complete all the testing cycles. It also takes into consideration the time when no testing is done and only the developers are working.
- Total MP Cost: defined as the total man power cost. This is calculated multiplying the number of work days by the number of people (testers and developers).
- Total Idle Cost: defined as the cost of time when the developers and tester were idle.
- Total Real MP Cost: defined as the cost of time when developers and testers were working.
- Ratio idle/total: Total idle cost/Total Cost

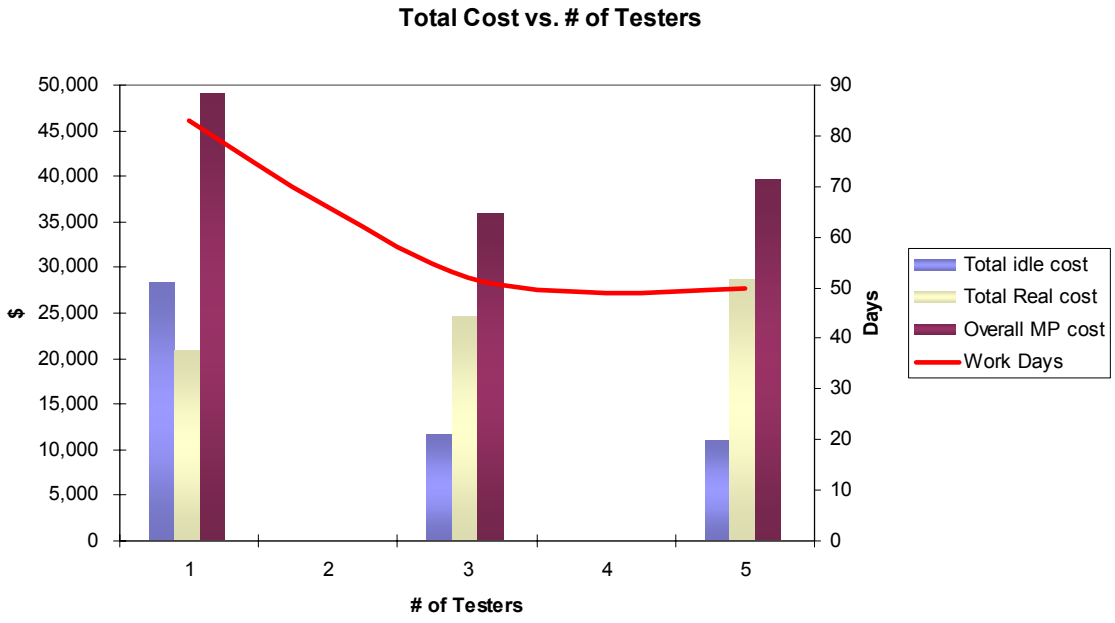

Figure 34. Cost vs. Number of Testers

Fig. 34 shows the results after trying different values for the number of tester. We observe that having more than 3 testers doesn't reduce the duration of testing phase. Conversely, having just one tester increases the idle cost (i.e. developers waiting for errors to be fixed).

Fig. 35 shows that increasing the maximum number of days per cycle over 10 days reduces both the duration of the testing phase and the overall MP cost.

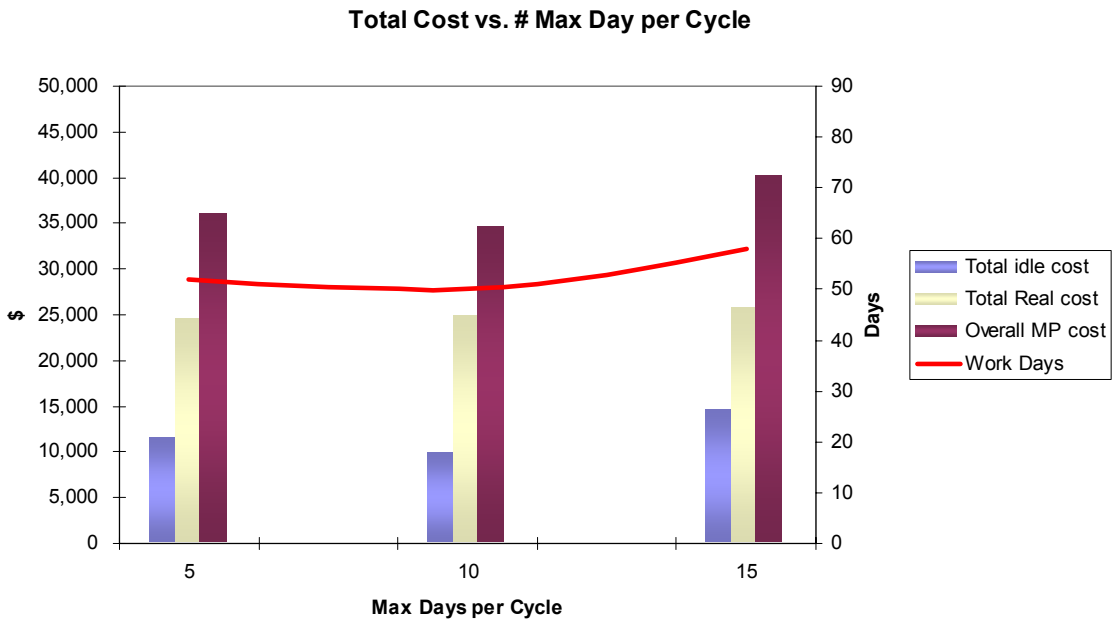

Figure 35. Costs vs. Max Day per Cycle

Analyzing the behavior of two variables combined (max number of errors and max number of days per cycle) we observe that the optimal combination is 30 errors and 10 days (see Fig. 36). This seems to indicate that increasing the duration of cycling times could reduce the duration of the phase.

Our simulations indicated that current setting of Company A (a maximum number of 30 errors or five days per testing cycle, 2.5 tester and 6 developers) are close to optimum values yet it seems to be space for some improvement. Increasing the maximum number of errors per cycle could decrease the Total Time MP Cost. Similarly, increasing the number of developer reduces the duration of the testing phase but increases the cost.

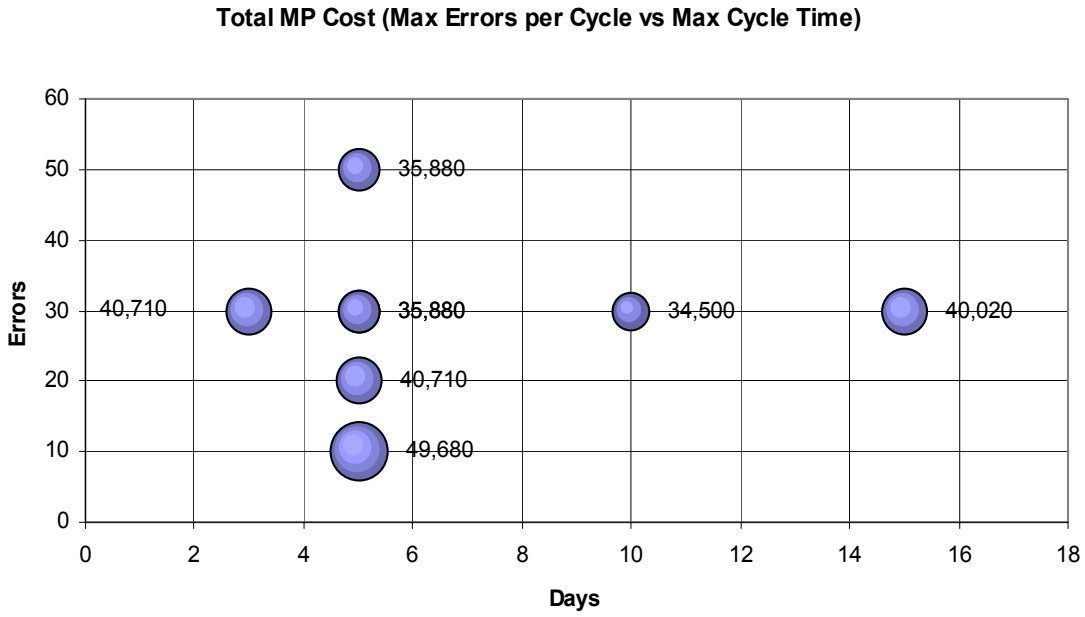

Figure 36. Total MP Cost (Max Errors and Max Cycle Time)