AMERICAN UNIVERSITY OF BEIRUT

PLAN-BASED VERSUS AGILE SOFTWARE DEVELOPMENT: A QUANTITATIVE APPROACH

by
HIBA JAMAL ITANI

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Engineering Management
to the Engineering Management Program
of the Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
April 2015

AMERICAN UNIVERSITY OF BEIRUT

PLAN-BASED VERSUS AGILE SOFTWARE DEVELOPMENT: A QUANTITATIVE APPROACH

by
HIBA JAMAL ITANI

Approved by:

[illegible]

Dr. Ali Yassine, Professor — Advisor
Engineering Management Program

[illegible]

Dr. Bacel Maddah, Associate Professor — Member of Committee
Engineering Management Program

[illegible]

Dr. Fadi Zaraket, Assistant Professor — Member of Committee
Department of Electrical and Computer Engineering

Date of thesis defense: April 27, 2015

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name:	Itani	Hiba	Jamal
	Last	First	Middle

[✓ Master's Thesis] ○ Master's Project ○ Doctoral Dissertation

[✓] I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, **three years after the date of submitting my thesis, dissertation, or project,** to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

هبه العيتاني

May 8, 2015

ACKNOWLEDGMENTS

My work wouldn’t have been possible hadn’t it been for the blessings that Allah has given me and the people He has put in my way.

I would like to express my sincere gratitude to my thesis advisor, Dr. Ali Yassine, for his continuous advice and support throughout my research process. I have learned so much from you.

I would like to thank Dr. Bacel Maddah for his help and for always being an inspiration. His interesting insights are among the main reasons I pursued this research. I would also like to thank Dr. Fadi Zaraket for his valuable input and professional advice.

Last but not least, I want to express my deep appreciation and gratitude to family, especially my parents, who have always supported me and sacrificed much of their time, effort, and resources for my sake.

AN ABSTRACT OF THE THESIS OF

Hiba Itani for Master of Engineering Management
Major: Engineering Management

Title: Plan-based versus Agile Software Development: A Quantitative Approach

A major decision that can influence the success of a project is the software development method used, which is the structure imposed on the software development process. Qualitative research suggests that neither plan-based methods nor agile methods are optimal for all types of projects. However, quantitative research on this topic is scarce.

In this study, we propose a model that compares the structures of the Waterfall and Scrum software development methods taking into consideration factors such as project size, team size, and requirement volatility. This model aims to aid in choosing the software development method that minimizes effort based on the characteristics of the project. Our results indicate that for highly volatile projects and large projects, Scrum is better than Waterfall, while for projects with low volatility and small projects, Waterfall is more desirable.”

CONTENTS

ILLUSTRATIONS

TABLES

Table Page

CHAPTER I

INTRODUCTION

With the increase in software complexity, software projects are often exceeding time and budget constraints, and suffering from ineffective coding and difficulties in software maintenance. The term “Software Crisis” was coined in the first NATO Software Engineering Conference in 1968 to refer to these problems. As a result of the software crisis, the processes and tools to design and build software efficiently became an area of high concern, and the field of software engineering emerged (O'Regan, 2008).

Efforts to improve software development have been helpful so far; however, the success rates of software projects are still not satisfactory. According to the CHAOS report, the percentage of projects that are delivered on time, on budget, and with the required features and functions has increased from 29% in year 2004 to 39% in year 2012 (Standish Group, 2013). While some factors that lead to project failure are uncontrollable, sometimes informed managerial decisions can make all the difference. A major decision that can influence the success of a project is the software development model, which is the structure imposed on the software development process.

Traditionally, the software development process had been a plan-based Waterfall model where software is developed in sequential phases (Royce, 1970). That is, the requirements of the project are negotiated and agreed on. Then, specifications that formalize the requirements are set and a design for the system is put in place. Based on that, the system is implemented through a series of refinement activities that include coding, unit testing, and integration of the different code units. Then, verification, functional system testing and acceptance testing take place. Testing might result in design and code changes causing a cycle in the process. Finally, the software is released and maintained as needed.

This systematic approach faced major problems especially when the requirements of software were susceptible to change or when the product requirements could not be fully agreed on initially. Other processes such as the V-model introduced variations to the traditional waterfall model by trying to better account for change management. However, the need for more flexible methodologies was still present, which gave rise to iterative and agile methods.

Iterative methods such as the Spiral Model introduced by Boehm (Boehm, 1988) and the Rational Unified Process (RUP) introduced by IBM (Eeles and Houston, 2002) aim to reduce risk through developing software in smaller portions. While iterative methods are not as rigid as traditional methods, they entail that requirements elicitation and analysis activities are to be done before the implementation of the project begins. For this reason, these methods are still classified as big-design-up-front or plan-based methods. Agile methods, on the other hand, follow an incremental and iterative software process where the different phases of software development including requirements analysis are interleaved in order to accommodate changing requirements, encourage customer involvement and create opportunities for learning and improvement.

In 2001, a group of seventeen leading software practitioners came together to discuss the problems with existing software methodologies and wrote “The Agile

Manifesto" that includes a description of the values that support agile or lightweight software development. Based on the values and principles set by the agile manifesto, a distinction between agile and non-agile practices emerged, and more focus was given to implement agile practices in software development.

Agile practices were able to find several supporters among software practitioners and with time, agile practices increased in popularity in the software industry. According to the "8th Annual State of Agile Survey", 52% of the 3,501 software developers surveyed (mostly from North America and Europe) said that they are using agile to manage the majority of their projects (APLN, 2013). Agile methods have joined the mainstream of development approaches and even large companies including HP, IBM, Oracle, and Microsoft are using agile methods (Moniruzzaman and Hossain, 2013).

So does that mean that agile methodologies are deemed automatically better than plan-based ones? The short answer for this question is no. Neither agile nor plan-driven methods represent a methodological silver bullet; however, one software process can be better than the other under different circumstances (Lindvall et al., 2002). Although a lot of research has been conducted to better understand the circumstances under which agile methods are preferable, most of this research uses expert judgment or empirical methods. While these approaches have their advantages, they have limitations when it comes to analyzing how the different factors influencing the software development processes do so, and how these factors interact with each other. Quantitative modeling could help overcome these limitations.

There are various plan-based and agile development approaches that are used among software practitioners today. Popular plan-based methods include the Waterfall model, the V-Model, the Spiral Model, and the Rational Unified Process (RUP). Popular agile methods include Scrum, Extreme Programming (XP), and Feature-Driven Development (FDD). While it is useful to compare plan-based approaches to agile ones on a general level, for a more specific and quantitative comparison, we have decided to focus on comparing Scrum to the traditional Waterfall method.

We aim to highlight the advantages and disadvantages of plan-based and agile methods by modeling the effort required for each of these processes under variable conditions and analyzing the results. For this purpose, we devised a model that compares a popular plan-based method, Waterfall, to a popular agile method, Scrum. We took into account the size of the project, the structure of the method used, and effects of requirements volatility. Our results indicate that Scrum is better under high volatility conditions while Waterfall is better when there is no or very low requirement volatility. Moreover, our model indicates that Scrum is more desirable for bigger projects.

CHAPTER II

BACKGROUND

This section includes a brief overview of software engineering history, a description of popular plan-based methods, and a description of popular agile methods. This background is important for understanding the context of this research.

A. Brief Overview of Software Engineering History

“Waterfall” and “Scrum” development methods are not ancient concepts; rather they are the results of the evolution of the field of software engineering over the past few decades. During the 1960’s the main approach towards software development was the “code and fix” approach where coding and testing happen with very little planning and design effort. However, software complexity was posing challenges and many projects were running over time and budget constraints and facing severe problems concerning efficiency and quality. This led the NATO Science Committee to hold two conferences in the 1968 and 1969 to address the situation. The conclusions of these conferences encouraged the emergence of more organized methods and formal practices in software management (Boehm, 2006).

During the 1970s, a more structured approach towards software development was becoming dominant. Software practices included software development practices where requirements and design phases preceded coding and testing. The 1970s decade witnessed the rise of the Waterfall formal method where the system is designed, the code is written and tested, and the project is maintained all in a sequential manner.

Despite the big leap in software engineering that formal methods provided, there was still a lot to be done regarding improving software productivity and processes. During the 1980s and 1990s, a lot of effort was invested in understanding the factors effecting software development productivity such as staffing, prototyping, and process improvement. Towards the end the twentieth century and the beginning of the twenty first, Waterfall and similar approaches were facing difficulties in adapting to rapidly changing requirements and iterative approaches with less up-front planning were becoming popular as opposed to the big-design-up-front traditional methods such as the Waterfall method. In 2001, a clear distinction between plan-based and agile methods was made through the set of values identified by the "Agile Manifesto". After the agile manifesto, not only did existing agile methods gain more popularity, but also new agile methods started emerging.

Today, both traditional methods and agile methods are being used in the software development industry, and the challenge lies in knowing which development method is better suited for a given software development project.

B. Plan-based Methods

There are several plan-based methods used the software industry. The most basic plan-based method is the Waterfall Model. After the Waterfall model was introduced there were some variations of it such as the V-model. These models are known as traditional plan-based models. Moreover, iterative plan-based methods such as the Spiral model and the Rational Unified Process (RUP) are also popular in the software industry.

1. Waterfall Model

"Waterfall" is the word used to describe the sequential software development model suggested by Royce in 1970 (Royce, 1970). The model divides activities of software development into distinct phases that should be completed in a stage-wise fashion. These phases are: system requirements, software requirements, analysis, program design, coding, testing, and operations. Figure 1 represents the waterfall model (Royce, 1970).

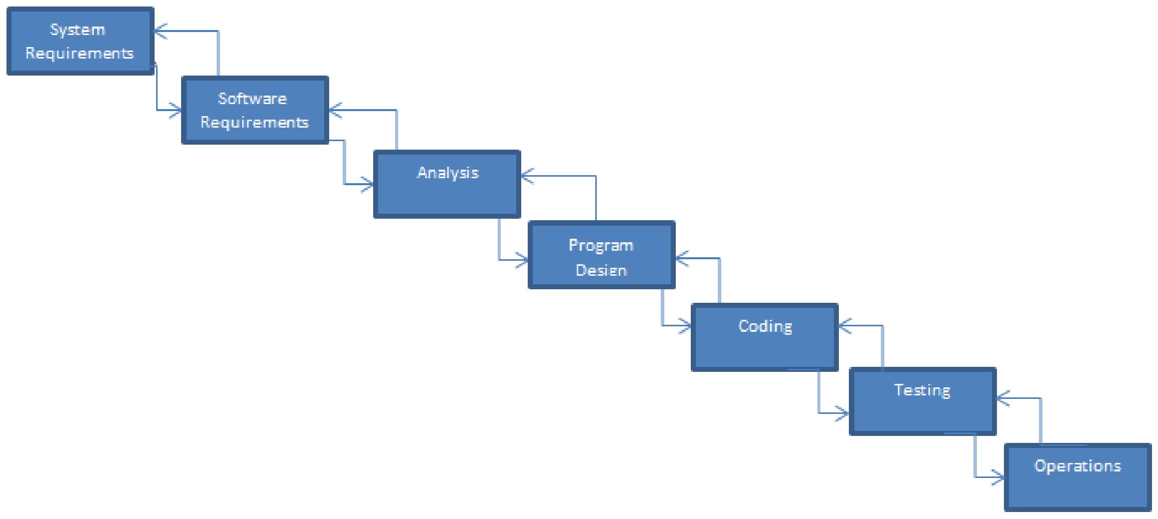


Figure 1: Waterfall Model

The project starts with defining system requirements which include components of the system such as required hardware and software tools. The next step is defining software requirements or the functionality that the software is expected to deliver. These requirements are usually documented and could be used as a contract between the customer and the entity responsible for developing the software. Analysis includes understanding the requirements and determining how they interact with each other and other applications like databases and user interfaces. Implementation of the system begins by program design which consists of high level architectural design and a detailed design for the different software components. After design is finalized, the coding phase begins and software is developed based on system specification and design. To verify and validate the system, a series of tests should be performed to check if the system meets the functional and non-functional requirements and to find and fix any errors in the system. Finally, the system is released and maintained in the operations phase of the Waterfall model.

Modified waterfall models have used the same sequential activities identified by Royce’s model but represented in a different way. For example, some models consider software requirements and analysis as one phase and architectural and detailed design as two separate phases (Mohammed et al., 2010). Moreover, the different phases of the waterfall are separate in the sense that one phase has to be finalized before the next phase begins. However, there could be feedback from current phases to older ones. For example, errors revealed in testing require moving back to the coding phase. While all of these phases are linked together, they are usually the responsibility of more than one entity. That is to say that a customer or an analyst can identify the requirements, while design could be done by software designers and code and testing by software developers.

2. V-shaped Model

The V-shaped model is another traditional approach to software development sometimes considered as an extension to the Waterfall Model. This model also consists of sequential phases; however more emphasis is placed on system validation and verification. The testing phase of the Waterfall Model is made more explicit, and the relationships between testing and other phases of the model are clearly identified (Rowen, 1990). Testing includes unit testing for the different components of the system, interface testing to check that these components interact with each other as planned, system testing to verify that the non-functional requirements such as security and reliability are met, and acceptance testing to make sure that the system performs as the customers want it to. Figure 2 is a representation of the V-model (Easterbrook, 2001).

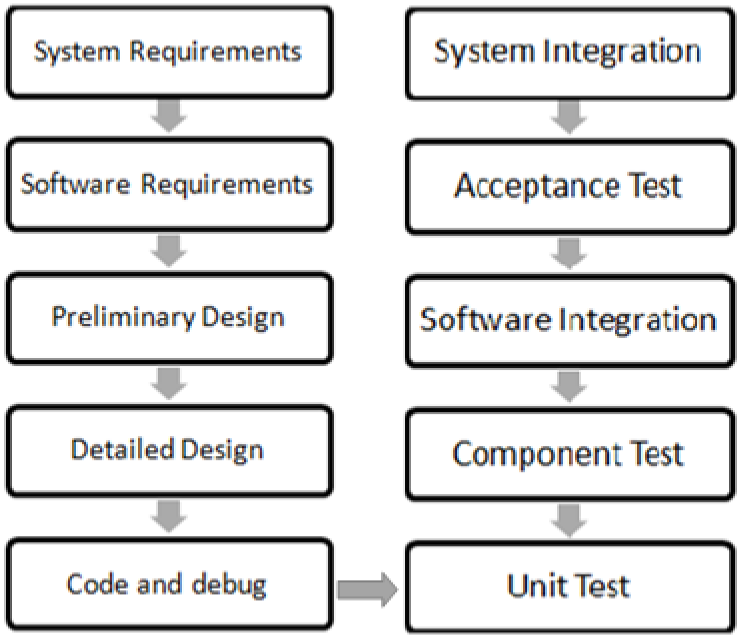


Figure 2: V-Model for Software Development

3. Iterative Plan-based Models

Iterative plan-based models are considered non-agile because they depend on big-design-up-front, unlike agile methods which don't allocate a lot of resources for design before development. Two of the most popular iterative plan-based methods, the Spiral model and the Rational Unified Process (RUP), are discussed in the Appendix.

C. Agile Methods

Agile methods are described as methods that adhere to the values and principles of the "Agile Manifesto". In this section, we give a brief overview of the Agile Manifesto and agile methods in general, and we describe one of the most widely used agile methods, Scrum.

1. The Agile Manifesto

Towards the end of the twentieth century, heavyweight traditional approaches to software development received serious criticism among software practitioners and lightweight software development methods started emerging. These lightweight, or agile, approaches focused on releasing portions of working software instead of following a plan-based approach with big design up-front.

On February 11-13, 2001, seventeen professionals representing supporters of different lightweight methodologies met at a ski resort in Utah. Their meeting resulted in what is now a milestone in the world of software development: The formation of the "Agile

Alliance” and the emergence of “The Agile Manifesto”. The Agile Alliance members identified the core values that agile methods are based on. Accordingly, The Agile Manifesto states (Alliance, 2001):

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

***Individuals and interactions** over processes and tools*

***Working software** over comprehensive documentation*

***Customer collaboration** over contract negotiation*

***Responding to change** over following a plan*

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

Moreover, this group also set principles behind the agile manifesto to guide agile practices. The twelve principles of Agile Software are listed in the Appendix.

While there is no clear definition for the term “Agile Development”, there is consensus that the values and principles suggested by the agile manifesto present guidelines for agility in software development. Based on that, several software development methods have been marked as agile, and although these methods have different processes and activities, they share the same values and principles stated above.

According to the 8th annual state of agile survey and Forrester Inc Survey of Agile methodologies used, the most popular agile methods include Scrum, extreme programming, and Feature Driven Development (APLN, 2013), (West et al., 2010). We will provide a description of the Scrum methodology which was listed as the most popular agile method with 73% of the respondents asked about agile methods used answering that it is Scrum or a Scrum variant. A description of Extreme Programming and Feature Driven Development is present in the Appendix.

2. Scrum

Scrum is a structured agile method with a focus on the framework used to manage complex software development (Sutherland and Schaber, 2013). That is to say, many of the practices identified in XP can be used in Scrum as long as the framework set by Scrum is not violated.

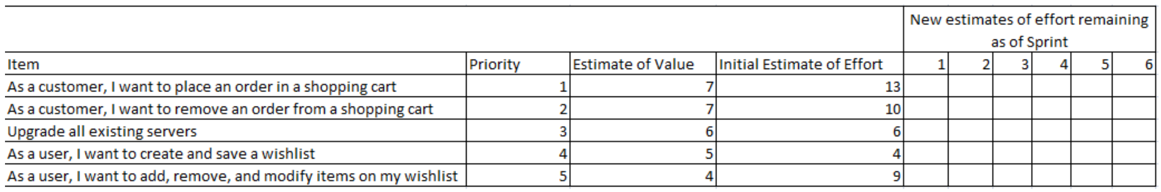
Scrum implements an iterative and incremental approach to increase predictability and control over risk. This iterative process happens in cycles called sprints where there is time for planning before each sprint, time for development, time for presenting a working demo of the functional product, and time for reflecting and learning from each sprint.

The three pillars that scrum aims for are transparency, inspection, and adaptation. The Scrum framework is structured to accommodate those pillars though identifying roles for the Scrum team members, tracking Scrum artifacts, and setting time-boxed Scrum events.

There are three main roles in every scrum team: a product owner responsible for maximizing the value of the product, a development team which consists of seven (plus or minus two) dedicated members who are responsible for a big part of the planning and for implementing the plan in increments, and a Scrum master which is responsible for ensuring that Scrum practices and rules are being followed.

Scrum identifies artifacts that help in tracking progress and explaining work flow. These artifacts are the product backlog, the sprint backlog, and the product increment.

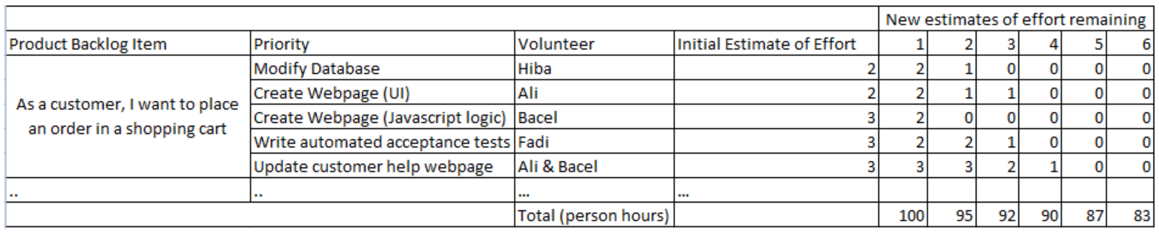
The product backlog represents a list of requirements that need to be present in the final product ordered by priority. The product owner identifies the items on the list and the way they are ordered based on what would maximize product value. This list is highly flexible and evolves over time to allow for changing requirements and modifying scope. Details like a description and an estimate of the items on the backlog is also present although high priority items on the top of the list are usually more detailed than lower priority items. While the product owner is responsible for this artifact, the Scrum master and members of the development team regularly help the product owner manage the product backlog through a process called backlog grooming where items on the list can be reordered, estimated, modified, or even added. Figure 3 shows an example of a product backlog (Sutherland, 2010).



				New estimates of effort remaining as of Sprint					
Item	Priority	Estimate of Value	Initial Estimate of Effort	1	2	3	4	5	6
As a customer, I want to place an order in a shopping cart	1	7	13						
As a customer, I want to remove an order from a shopping cart	2	7	10						
Upgrade all existing servers	3	6	6						
As a user, I want to create and save a wishlist	4	5	4						
As a user, I want to add, remove, and modify items on my wishlist	5	4	9						

Figure 3: Product Backlog used in the Scrum Process

The sprint backlog is a subset of the product backlog which includes the requirements that should be achieved by the end of the coming sprint. The sprint backlog is therefore the functionality that is expected to be delivered by the development team at the end of a sprint. The sprint backlog is managed by the development team during the planning and development phases. It has much more detail than the product backlog as it presents a clear idea of the process followed and can be updated on a daily basis to show the work done so far and the work remaining. Figure 4 shows an example of a sprint backlog (Sutherland, 2010). As shown in the example, each sprint task is assigned a member or volunteer and has an estimated effort which could be updated on a daily basis.



				New estimates of effort remaining					
Product Backlog Item	Priority	Volunteer	Initial Estimate of Effort	1	2	3	4	5	6
As a customer, I want to place an order in a shopping cart	Modify Database	Hiba	2	2	1	0	0	0	0
	Create Webpage (UI)	Ali	2	2	1	1	0	0	0
	Create Webpage (Javascript logic)	Bacel	3	2	0	0	0	0	0
	Write automated acceptance tests	Fadi	3	2	2	1	0	0	0
	Update customer help webpage	Ali & Bacel	3	3	3	2	1	0	0
..	..	...	...						
		Total (person hours)		100	95	92	90	87	83

Figure 4: Sprint Backlog used in the Scrum Process

The sprint backlog helps the team achieve transparency which is essential for tracking and managing progress. An effective tool used by Scrum teams to track progress is the sprint burndown chart which visually shows progress as a function of time. Figure 5 shows an example of a sprint burndown chart (Sutherland, 2010). According to the figure below, the team seems to be behind schedule and the speed of development is lower than expected. The technical term used to refer to the speed of development is the velocity of the team. In this example, the team should improve its velocity in order to finish the required tasks on time. Similar tools can be used to measure the progress of the project or a portion of the project known as a release.

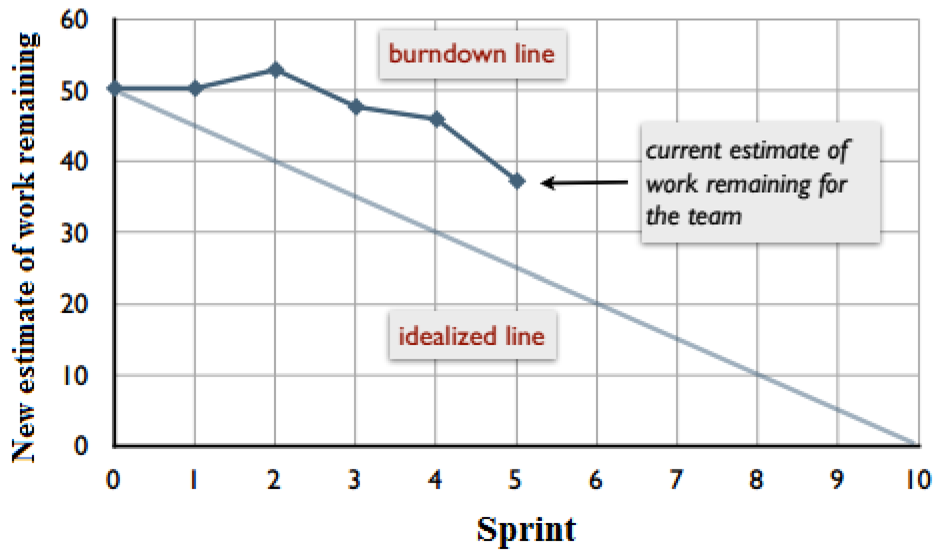


Figure 5: Sprint Burndown Chart (Sutherland, 2010)

Based on the product backlog and work achieved after each sprint, the product increment is delivered. The product increment is simply an aggregate of all the functional product backlog items developed during the last sprint and all previous sprints. Since the Scrum team works on achieving shippable functionality by the end of each sprint, the product increment is the part of the product which can be released for use upon the request of the product owner.

After understanding the different roles and Scrum artifacts, the workflow of the Scrum framework can be explained through the well-defined and time-boxed Scrum events. The main events in each sprint are: The sprint planning meeting, the daily Scrum, the sprint review, and the sprint retrospective.

Before each sprint, a sprint planning meeting takes place. This meeting is set to be eight hours for a one month sprint or 4 hours for a two week sprint. The meeting consists of two main parts. The first part of the meeting is dedicated to agree on the items of the product backlog that are to be included in the sprint backlog, or in other words, to agree on the requirements that would be delivered at the end of the sprint. The second part of the meeting revolves around how the development team intends to accomplish the functionality by the end of the sprint. This includes designing the system and giving detailed estimates for the tasks to be completed.

After the sprint planning meeting, the actual development of the system happens during the sprint which is typically one week to four weeks long. During the sprint, a daily Scrum standup meeting takes place where all the development team, in addition to the Scrum master, meet for fifteen minutes at the beginning of each day to update each other about their progress since the last meeting, what they plan to do before the next meeting, and any obstacles that they are facing. The daily Scrum helps in improving communication, discovering and solving problems, and removes the need for managerial meetings during development.

If all goes according to plan, by the end of the sprint, the development team should have a working demo with the functionality agreed on. The product increment is shown to the product owner and possibly other stakeholders at the sprint review meeting which is time-boxed to four-hours for a four week sprint and two-hours for a two week sprint. Based on that meeting, the product owner gives feedback, identifies what has been done and what needs more work, and cooperates with the development team to agree on what should be done next. The meeting results in a revised backlog and valuable input for the next sprint planning meeting.

Last but not least, before a new sprint begins and the cycle repeats, a sprint retrospective meeting is held to reflect on how the sprint went and suggest improvements to the development process accordingly. This meeting is time-boxed to three hours for a one month sprint and shorter if the sprint is shorter. Figure 6 shows a summary of the main Scrum concepts that we have discussed (Sutherland, 2010).

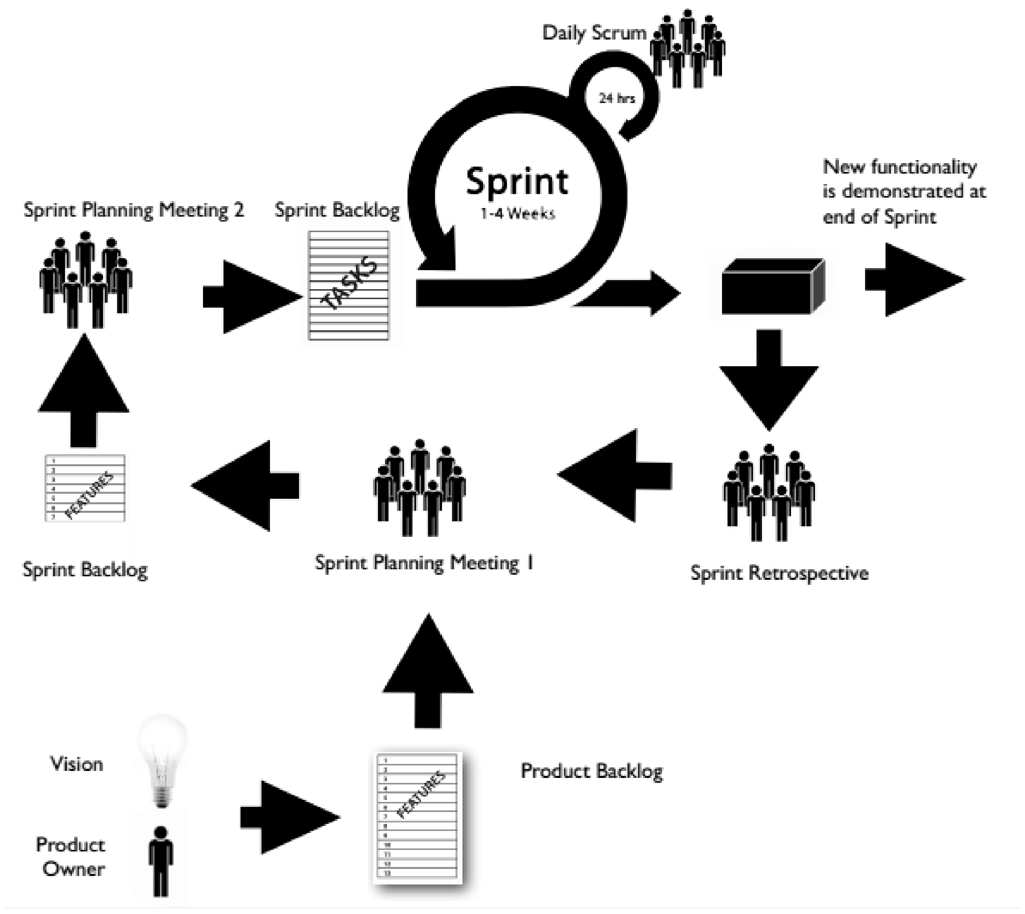


Figure 6: The Scrum Development Process (Sutherland, 2010)

In case of large-scale agile development, Larman suggested two Scrum frameworks that scale up the Scrum process in an effective way. These frameworks are very similar to the Standard Scrum method discussed above with few differences. Framework-1 is applied for team sizes larger than ten but less than one hundred. This framework is centered around the idea of splitting the developers in teams, where each team has a Scrum master and product owner. These teams coordinate with each other before, during, and after each sprint. If the number teams exceeds ten, framework-2 is applied. Framework-2 splits the teams in different areas, where the teams in each area are specialized in one requirement area. The different areas as well as the different teams coordinate with each other and with the product owners before, during, and after each sprint (Larman and Vodde, 2013).

CHAPTER III

LITERATURE REVIEW

This section gives an overview of popular Software Engineering methods with special focus on Waterfall and Scrum. It also includes a list of advantages and disadvantages for using agile methods and a general idea about relevant research done in this area.

A. Quantifying Project Success

The aim of project management is the success of the project. Studies found that the attributes of success for a project are quality, scope (meeting all customer requirements), and meeting time and cost constraints (Cohn and Ford, 2003), (Lindvall et al., 2004).

Software quality is defined by the quality models created by McCall and Boehm (McCall et al., 1977), (Boehm et al., 1978). These models identify measurable software quality factors such as correctness, reliability, efficiency, portability, flexibility and others. They also identify how these metrics are measurable. For example, flexibility is defined by McCall as “the effort required to modify an operational program” (McCall et al., 1977).

Scope refers to the requirements set by the customer. These requirements can undergo changes, additions, or deletions. Changes to requirements are known as scope churn, requirement additions are known as scope creep, and deletion of requirements is known as scope scrap (Kulk and Verhoef, 2008). To achieve project success, the project must meet the customer requirements after it undergoes requirements churn, requirements scrap, and requirements creep. Certain guidelines can be specified ahead of time to limit changes in scope.

Although meeting all the requirements and having a good quality product are essential factors, these are measures of produce performance. In order to assess the success of a project process performance such as time and cost should be also taken into account (Shao et al., 2014). Although both plan-based and agile methods promise to deliver a good quality product meeting all requirements, a major concern remains regarding how effective these methods are in meeting schedule and budget constraints. While timeliness could be measured as the overall time it takes to finish a project (in hours or months), quantifying cost is not as straight forward. Cost can include monetary as well as other resources such as human resources.

While it is hard to optimize all of these factors at the same time, it is worth noting that these factors are not independent of each other. For example, Harter et. al. found that product quality is usually better when the time spent on the project is also reduced (Harter et al., 2000). Moreover, some metrics can be used to reflect more than one factor of project performance. Effort, often measured in man-hours, is a measure that combines two aspects: time and cost.

Nevertheless, sometimes success factors collide and compromises have to be made depending on customer priorities. That is to say, we can demand that certain scope or quality constraints are met and try to minimize the cost of the project accordingly.

B. Software Size Estimation

The effort, cost, budget, and time required to develop a software product depend on the size of the project. Therefore, software size estimation metrics should be investigated to understand how software size could be quantified. Software size is used to quantify software length, functionality, and complexity (Bajwa, 2009). Popular size metrics include lines of code, use case points, function points, and story points.

1. Lines of Code

One way to measure software size is by counting the lines of code. Although this metric is widely used, it is not simple. When using lines of code as a metric for software size, it is important to define exactly what is being measured. There are many variations for lines of code. For example, there is some ambiguity as to if lines of code include data definitions, comments, and job control language, and whether physical or instructional lines of code should be measured (Kan, 2002). Although lines of code are easy to count once a definition is set, they do not reflect the functionality or complexity of the system (Bajwa, 2009).

2. Function Point Analysis (FPA)

Function points are a very popular method to measure the size of the system by measuring its functionality. Function points relate to five software components: Number of user inputs, number of user outputs, number of user inquiries, number of internal logical files, and number of external interfaces. Software requirements are classified into these five categories and are then weighted according to the complexity of the software to obtain the unadjusted function points. The function points are then adjusted to include the factors captured the Value Adjusted Factor (VAF) (Cheung et al., 1999).

The Value Adjusted Factor adjusts the unadjusted function point count by up to 35% and is based on the degree of influence of the 14 General System Characteristics (GSC) which are: Data communications, distributed data processing, performance, heavily used configuration, transaction rate, online data entry, end-user efficiency, online update, complex processing, reusability, installation ease, operational ease, multiple sites, and facilitate change (Bundschuh and Dekkers, 2008).

Function point analysis well documented and has set standards for calculation. Unlike lines of code, it can be calculated during the early phases of a project. A disadvantage of the function point method is that it is hard and sometimes costly to calculate especially if the project is big.

3. Use Case Points (UCP)

A use case can be defined as a list of steps or a description of the sequences of interactions possible between the system and other actors within the scope of a certain goal (Adolph et al., 2002). In object oriented programming, use cases are often used as a tool to model the functional requirements of a system. Influenced by the function point method, use case points are another measure of size based on functionality.

The first step for calculating the use case points of a system given its use cases is to calculate the unadjusted actor weight. This is done by categorizing the actors in the use case model as simple actors (other systems with defined application programming interfaces), average actors (other systems interacting through internet protocols), and complex actors (such as a person interacting through a graphical user interface). By counting the number of actors in each category and multiplying the number of simple actors by 1, the number of average actors by 2, and the number of complex actors by 3, we get the *unadjusted actor weight* (Anda et al., 2001).

The second step is to find the *unadjusted use case weight* by classifying use cases into simple, average, and complex depending on the number of transactions in the use cases. A use case with 3 or less transactions is multiplied by 5, a use case with 4 to 7 transactions is multiplied by 10, and a use case that has more than 7 transactions is multiplied by 15. The unadjusted use case weight is then added to the unadjusted actor weight to obtain the *unadjusted use case points*.

These unadjusted use case points are then adjusted to reflect technical and environmental factors (Anda et al., 2001). Figure 7 shows a summary of the method used to find use case points.

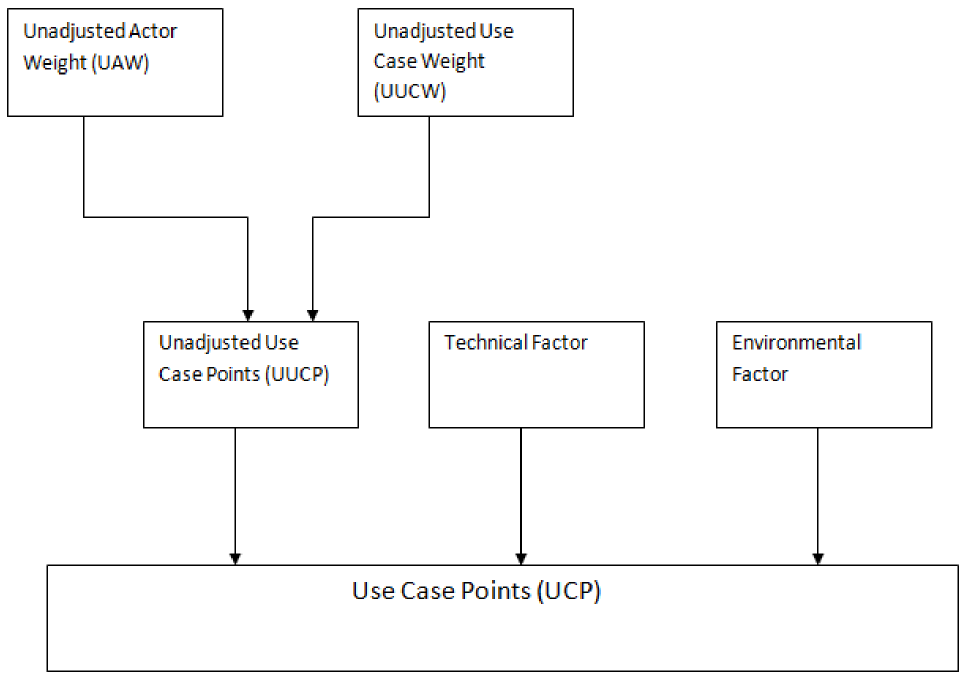

Unadjusted Actor Weight (UAW)
Unadjusted Use Case Weight (UUCW)
Unadjusted Use Case Points (UUCP)
Technical Factor
Environmental Factor
Use Case Points (UCP)

Figure 7: Calculating Use Case Points

Use case points are a good indicator of functionality, complexity and effort, and they can be calculated early on in the project. However, they require having a set of use cases which is not always available especially in agile projects.

4. Story Points

Story points are used by Scrum teams to estimate the size of the functionality that will be implemented in the next sprint. Requirements identified in the product backlog are rewritten in the form of user stories. User stories describe a set of functionalities that is independent, negotiable, valuable to users and customers, estimable, small, and testable (Cohn, 2004). The scrum team assigns story points to user stories by a paired comparison process. Based on how much story points the team expects to be able to complete in one sprint, the team then selects which user stories to implement. This process is not standardized and differs from one team to another (Felhman and Santillo, 2010). Therefore, the story point method is often criticized for being highly subjective.

C. Software Cost Estimation Methods

Several software estimation methods have been developed to estimate the time, cost, and effort required to complete a software project. These methods include estimation through algorithmic models, expert judgment, and estimation by analogy (Boehm, 2007). Our analytical model uses some of the results obtained by using these methods in order to make reasonable assumptions.

1. Algorithmic models

There are several algorithmic models used for software estimation purposes like the Constructive Cost Model (COCOMO), the Constructive Systems Engineering Model (COSYSMO), Software Lifecycle Management (SLIM), and Function Point Analysis (FPA) (Shepperd et al., 1996). These models provide estimates for cost based on a set of variables or cost drivers (Boehm, 2007).

For example, the COCOMO model is based on an empirical study of 63 projects. The basic model estimates development effort and time given the size of the project and the mode of development. Intermediate and advanced versions of the model exist to account for cost drivers such as product, hardware, personnel, and project attributes across different phases of the project (Merlo–Schett et. al, 2003).

According to the USC COCOMO reference manual (Horowitz, 1994), the development time and effort can be estimated for Waterfall projects for which the constraints are somewhat flexible as:

$$Development\ Effort = 3\ \times (Project\ Size)^{1.12}$$

$$Development\ Time = 2.5\ \times (Development\ Effort)^{0.35}$$

Note that the project size used by COCOMO is expressed in thousands of delivered code instruction. Moreover, the number of developers required to finish the project is expressed as:

$$N = \frac{Development\ Effort}{Development\ Time}$$

2. Analytical Models

There are several models that use an analytical method to gain interesting insights about software engineering. In our model, we aim to compare traditional methods to agile methods by formally modeling these methodologies. That will allow for deep analysis of these models similar to how modeling certain aspects of software development has allowed others to come up with interesting insights.

One model that uses analytical methods to come up with insightful results regarding iterative development is the model suggested by Koushik and Mookerjee (Koushik and Mookerjee, 1995). This model finds the optimal size of a team and the optimal number of modules to be integrated in one iteration in order to minimize the effort needed for coordination activities. They show that as the time available decreases, the level of coordination decreases and the team size increases. They also show that with large teams, integrating more modules per iteration is better, and that as the system size increases, more coordination is required.

Another quantitative model that offers valuable insight is suggested by Jansi and Rajeswari. Jansi and Rajeswari suggested an approach for sprint planning in agile methods based on an integer planning model. Their model aims to maximize utility by choosing optimal number of stories to be included in each sprint (Jansi and Rajeswari, 2015).

A study on organizing knowledge workforce for specified iterative software development tasks minimizes the time needed for a project subject to budget constraints by assigning the right tasks to the right people. Their model results in interesting insights on allocation of resources for an iterative software project (Shao et al., 2014).

3. Expert Judgment

Expert judgment as name implies is the process of consulting experts to estimate the cost, time, or effort that a project may take. This can be done informally, or through a formal technique like the Delphi method where the most reliable consensus of opinion is considered for a group of experts (Rowe and Wright, 1999). Although widely used, this method is subjective in nature and is not very useful in visualizing how sensitive the project is to changes in its characteristics.

4. Analogy

This method consists of describing the project in terms of variables and then finding projects similar to the project under study. Historic data about effort, cost, and time derived from for similar completed projects can be used to find estimates for the new project. This method can be troublesome especially in the stages of finding similar projects and evaluating the degree of similarity (Shepperd et al., 1996).

D. Comparative Literature

Traditional and agile methods have been compared with respect to many aspects. While most of the research on this topic is qualitative in nature, some quantitative research also exists. These methods differ from our model by the tools used and the aspects being compared.

1. Qualitative Comparison

According to Kumar and Bhatia, the benefits of the agile methodology over the traditional plan-based method include handling change of requirements since the customers are directly involved in the development process, fault detection as testing is performed frequently, increased performance with the help of daily meetings, flexibility of design, and improvement in quality. However, they also point out that the agile has limitation when compared to plan-based methods. The limitations of agile methods include not having enough focus on product design, having big managerial overhead, and needing a lot of coordination and communication (Kumar and Bhatia, 2012).

According to Awad, the number of developers needed by traditional methodologies is bigger than the number of developers required with agile methods especially with large projects, and traditional methods are more effective for larger teams than agile methods. Moreover, he points out that heavyweight methods involve many activities that lead to longer time until delivery such as documentation, design documents, and writing analysis, which means that for tight deadlines, agile is preferred over traditional methods (Awad, 2005).

According to Boehm and Turner, there is no method that can be labeled as the most suitable for all software projects. They argue that agile methods are good at handling changeability and invisibility due to constant communication and sharing, but do not handle complexity and conformity in an ideal way because they do not scale up very well, and they do not enforce much discipline and order in the workplace. Moreover, they mention that plan-driven methods are good at handling conformity and invisibility by investing in documentation, but they fail to handle changeability and complexity in a proper manner. Boehm and Turner also point out that iterative and waterfall methods have home grounds were one clearly dominates the other. Agile methods are more suited for projects that need to respond to change and turbulent environments, while the plan-based methods like the waterfall model are more suitable for predictable projects, large teams, stable environments, and situations where it is difficult for the customer to be dedicated on-site (Boehm and Turner, 2003).

Table 1 below summarizes some major differences between agile and traditional methods.

Table 1: Comparison between Plan-based and Agile Methods

Aspect	Plan-based Methods	Agile Methods
Software Development Lifecycle	Sequential (Moniruzzaman and Hossain, 2013)	Evolutionary; Iterative and incremental (Moniruzzaman and Hossain, 2013)
Customer Involvement	Only during the beginning of the project (Hoda et al., 2010)	Essential to the success of the project; Throughout the project (Ahmed, 2010)
Documentation	Comprehensive (Moniruzzaman and Hossain, 2013)	Light (Moniruzzaman and Hossain, 2013)
Development Team	Tightly controlled by project manager (Moniruzzaman and Hossain, 2013)	Self-directed (Moniruzzaman and Hossain, 2013)
Number of Developers	Large for large projects	Usually small even for large projects
Project Management Tasks	Schedule series of events, schedule people and resources, calculate critical paths, etc… (Dubakov and Stevens, 2008)	Product/Release backlog maintenance, burn down reports, task board, etc…(Dubakov and Stevens, 2008)
Popular Supporting Development Tools	MS project, Subversion, Bugzilla, etc…(CapTech, 2015)	XPlanner, ScrumWorks, Rally,VersionOne, Jira, etc… (CapTech, 2015)
Cost of Change	Exponential (Boehm et al., 2008)	Flat (Cockburn, 2000)
Change Attitude	Avoids change (Moniruzzaman and Hossain, 2013)	Welcomes change (Moniruzzaman and Hossain, 2013)
Knowledge Management	Explicit	Tacit
Upfront Planning	Heavy	Light

2. Quantitative Comparison

Quantitative comparison between plan-based and agile development is scarce. However, there are a few models in the literature that use quantitative tools to compare plan-based and agile methods.

For example, a system dynamics model based on the relationships between system variables is used to compare agile methods to the traditional plan-based method. The model describes the behavior of Scrum, Kanban, and Waterfall under similar starting conditions, and then compares agile methods to plan-based methods in terms of performance (Cocco et al., 2011).

Empirical research is also used to compare Waterfall to iterative models. According to the results of an empirical study on phase effort distribution data from the China Software Benchmarking Standard Group (CSBSG) database, iterative processes employ less effort percentage in plan and requirement, design, and test phases, and more effort in the coding phase (Yang et al., 2008).

CHAPTER IV

MODEL

In this chapter we list the assumptions we used for building our model. Then, we derive the model that takes into account the different structures of Waterfall and Scrum software development methods and the effects of requirements volatility.

A. Model Assumptions

Finding the best development method for a software project is a complex problem with many variables. In our approach, we narrowed down the problem by making several assumptions.

1. General Assumptions

First, we assume that the decision of the optimal software development method is limited to Waterfall and Scrum. In reality, the optimal development method can be neither. The reason for narrowing down our analysis to Waterfall and Scrum is that these two development methods are good representatives of the traditional and agile methods respectively. Moreover, we assume that both development methods are applicable to the project. Sometimes, Scrum or Waterfall is inapplicable like when customer involvement is infeasible or when the requirements cannot be known upfront. We also assume that in either case the project is always successful.

2. *Scrum Model Assumptions*

For Scrum, we assume that a sprint is two-weeks long, ten working days, and that the scrum team consists of three to ten developers, a scrum master, and a product owner which is a good practice for a Scrum project. Having less than three developers decreases interactions and productivity gains, while having more than ten developers requires too much coordination. For big projects, Scrum is scaled up by following Scrum of Scrums. Moreover, we assume that the first sprint does not include development activities, and that there is only one release: the final release, which is a reasonable assumption for a project that could be developed using the Waterfall method.

3. *Optimizing Effort*

We have decided to select the development method based on minimum effort required. Effort gives a good indication about the cost of the project as well as the time needed. Moreover, we assume that the project will be completed meeting the requirements for quality and scope which are reflected in the inputs of the model as the size of the project.

4. *Using Use Case Points*

For this model, we assume that the size of the project is measured by the number of use case points (UCPs). Although using UCPs as a size metric means that some additional effort might have to be done in order to form use cases and determine the UCPs, it can be deduced at the early stages of the project, and is applicable to both Waterfall and

Scrum as long as the development is object-oriented. We favored UCPs over function points and lines of code because function points are more costly. As for story points, they are a good measure for Scrum projects, but they are not applicable for Waterfall projects.

McKinsey & Company published an article in 2013 encouraging software practitioners to apply use cases and use case points as project metrics based on their own experience (Huskin et al., 2013). They argue that UCPs can be calculated in the early stages of a project and then modified as the project progresses, which makes them very useful for project planning even for agile projects.

Another supportive argument for the applicability of UCPs to agile projects is a case study of effort estimation in a project following agile software development using use case points. The study concludes that when UCPs were applied to an agile project, they produced estimates close to the actual effort spent on developing a project (Ani and Basri, 2013).

B. Proposed Model

The objective is to minimize effort by selecting the suitable software development process given the size of the project in UCP, the effort per UCP, and requirement volatility. Table 2 presents the parameters used in the model with their respective symbols.

Table 2: Parameters Used in the Model

Parameter	Symbol
Project original size (UCP)	S
Waterfall total effort (man-hours)	E_W
Scrum total effort (man-hours)	E_S

Required effort per UCP (man-hours)	E_{UCP}
Implementation effort per UCP (man-hours)	IE_{UCP}
Development effort per sprint for Scrum (man-hours)	DE_S
Total effort per sprint for Scrum (man-hours)	TE_S
Number of developers for Waterfall	N_W
Number of developers for Scrum	N_S
Number of teams for Scrum	D
Number of areas for Scrum	A
A binary indicator that is 1 in case there is more than one scrum team	b_1
A binary indicator that is 1 in case there is more than one scrum area	b_2
Waterfall total time (hours)	T_W
Scrum total time (hours)	T_S
Number of Sprints (Scrum)	SPR
Relative cost of change for Waterfall across the project lifecycle	C_W
Relative cost of change for Scrum across the project lifecycle	C_S
Average expected requirements volatility across the project lifecycle	RV

1. *Estimating Effort*

a. Waterfall

In case of no requirement volatility, the total Waterfall effort for the project is the size of the project in UCP times the needed effort to complete one UCP,

$$E_W = S \times E_{UCP}.$$

In the case of requirement volatility, we account for it by multiplying the total effort by a factor that accounts for both volatility and the cost of volatility,

$$E_W = S \times E_{UCP} \times (1 + Cost_w \times RV).$$

b. Standard Scrum

Estimating the effort needed for the scrum process can be done by estimating the number of sprints needed to complete the project and multiplying the number of sprints by the estimated effort per sprint. Since scrum can be scaled up by using the frameworks suggested by Scrum of Scrums, we will find the effort per sprint for a standard scrum team of 10 or less developers, for a group of ten or less scrum teams following Framework-1, and for a group of more than 10 teams following Framework-2 (Larman and Vodde, 2013).

For the case of no requirement volatility, we start by estimating the effort per sprint. A sprint includes a sprint planning meeting, daily Scrum meetings, a sprint review meeting, a sprint retrospective meeting, backlog grooming, and the actual implementation of the product. The sprint planning is time-boxed to four hours for a two-week sprint cycle. All the developers, in addition to the Scrum master, attend this meeting. Therefore the effort required for the sprint planning meeting is estimated to be $(N+1)$ people $\times$ 4 hours/sprint = $4\times (N+1)$ man-hours/sprint.

The daily Scrum meeting is time-boxed to 15 minutes and it is attended by the Scrum master and the developers. Considering that the sprint is 10 working days, and that the daily scrum is done every day except for the first and last day, the effort required for daily Scrum meetings is estimated to be $(N+1)$ people $\times$ 8 days $\times$ 0.25 hours/day = $2\times (N+1)$ man-hours/sprint.

Backlog grooming is the process of refining the product backlog. It usually happens during the sprint development time and is attended by the Scrum master and the development team. Backlog grooming is estimated to take five to ten percent of the sprint time (Sutherland, 2010). We estimate the effort needed for backlog grooming to be one working day or $8(N+1)$ man-hours/sprint, at 10% of sprint time.

The sprint review and sprint retrospective meetings are held at the end of the sprint and are estimated to take two hours each on average for a two-week sprint though this time can be a little less or more. The Scrum master and the developers are expected to participate in these meetings. Therefore, the effort required for these two meetings is estimated to be (N+1) people $\times$ 4 hours/sprint = 4$\times$ (N+1) man-hours/sprint.

For a two-week sprint, the time spent on development of the product is the time spent on other activities subtracted from the total time available. Assuming that there are five working days per week and eight hours per day, the total time available during a two-week sprint is 10×8= 80 hours. The time spent on sprint planning, daily Scrum meetings, and sprint review and retrospective meeting is 4 + 8 ×0.25 + 8 + 4= 18 hours/sprint. Therefore the time available for development is estimated to be 62 hours/sprint.

Knowing the implementation effort per UCP, and taking into account that the first sprint is spent on activities other than development such as high level design, we can find the number of sprints required as

$$SPR = \left\lceil \frac{(S \times IE_{UCP} \times (1 + Cost_S \times RV))}{DE_S} \right\rceil + 1$$

The total effort per sprint is therefore the sum of the effort required for all activities is

$$TE_S = 4 \times (N + 1) + 2 \times (N + 1) + 8 \times (N + 1) + 4 \times (N + 1) + 62 \times N$$

$$= 80 \times N + 18.$$

Table 3 summarizes the effort and time distribution within one sprint following Standard Scrum having a total of N developers.

Table 3: Effort and Time Distribution following Standard Scrum

Activity	Effort (in man-hours)	Time (in hours)
Sprint Planning	4(N+1)	4
Daily Scrum	2(N+1)	2
Product Backlog Refinement	8(N+1)	8
Sprint Review	2(N+1)	2
Team Retrospect	2(N+1)	2
Development	62N	80-18=62

The total effort for the project is

$$E_S = SPR \times TE_S = \left(\left\lceil \frac{(S \times IE_{UCP} \times (1 + Cost_S \times RV))}{62\,N} \right\rceil + 1\right) \times (80N + 18)).$$

TE_S is the total effort needed per sprint and can be deduced by summing up the efforts for the different activities during the sprint shown in Table 3.

c. Scrum of scrums: Framework-1

In the case of more than ten developers per team, the developers should be split into smaller teams that coordinate with each other, where each team has its own Scrum master (Larman and Vodde, 2013). These teams work in parallel in a way very similar to the typical scrum framework but that differs in the following aspects. First, the sprint planning meeting 1 which consists of two hours is performed jointly by all teams with two members from each team attending it. Second, there is an interteam coordination meeting that takes around 3 hours per sprint and is attended by a developer from each team. Product backlog refinement is performed by all developers and Scrum masters during the middle of a sprint and it takes around 8 hours. Moreover, the sprint review meeting is a common meeting attended by two developers from each team, and a joint retrospect meeting of about an hour and a half takes place at the beginning of the next sprint, and is attended by one developer from each team in addition to the Scrum masters. We assume that during the sprint review and interteam coordination meetings, the rest of the development team is working on implementation tasks.

Table 4 below summarizes the effort and time distribution within one sprint following Framework-1 having *D* number of teams and a total of *N* developers.

Table 4: Effort and Time Distribution following Framework-1

Activity	Effort (in man-hours)	Time (in hours)
Sprint Planning 1	4*D*	2
Sprint Planning 2	2(*N*+*D*)	2
Daily Scrum	2(*N*+*D*)	2
Interteam coordination	3*D*	3
Product Backlog Refinement	8(*N*+*D*)	8
Sprint Review	4*D*	2
Team Retrospect	2(*N*+*D*)	2
Joint Retrospect	3(*D*)	1.5
Development	60.5N - (3*D* + 4*D*) = 60.5*N* - 7*D*	80 - 19.5 = 60.5

In this case, $E_S = SPR \times TE_S = \left(\left\lceil \frac{(S \times IE_{UCP} \times (1+Cost_S \times RV))}{60.5N-7D} \right\rceil + 1\right) \times (74.5N + 21D))$.

d. Scrum of Scrums: Framework-2

In case the number of teams exceeds 10, another framework is suggested to help the team coordinate with each other (Larman and Vodde, 2013). Framework-2 is very similar to Framework-1 but it splits groups of teams into different areas. Each group of teams is responsible for a certain area. The same structure of framework-1 applies with the exception of a Sprint pre-planning meeting which occurs before every sprint for the overall product and expected to take around 2 hours, an additional two-hour Overall Sprint Review meeting, and an hour and a half Overall Sprint Retrospect meeting all of which are usually attended by two developers from each area. We denote by A the number of areas.

Table 5 summarizes the effort and time distribution within one sprint following framework-2 having D number of teams, A number of areas, and a total of N developers.

Table 5: Effort and Time Distribution following Framework-2

Activity	Effort (in man-hours)	Time (in hours)
Sprint Pre-Planning	$4A$	2
Sprint Planning 1	$4D$	2
Sprint Planning 2	$2(N + D)$	2
Daily Scrum	$2(N + D)$	2
Interteam coordination	$3D$	3
Product Backlog Refinement	$8(N + D)$	8
Product Sprint Review	$4A$	2
Sprint Review by Area	$4D$	2
Team Retrospect	$2(N + D)$	2
Joint Retrospect by Area	$3D$	1.5
Overall Joint retrospect	$3A$	1.5
Development	$59N - (3D + 4A + 4D) = 59N - 7D - 4A$	80 - 21=59

In this case, total effort is

$$E_S = SPR \times TE_S = \left(\left\lceil \frac{(S \times IE_{UCP} \times (1+Cost_S \times RV))}{59N-7D-4A} \right\rceil + 1\right) \times (73N + 21D + 7A).$$

e. A general form

Let b_1 and b_2 be two binary indicators. In case of standard scrum, b_1 and b_2 are zero. In case of Framework-1, $b_1 = 1$ and $b_2 = 0$, and in case of Framework-2, $b_1 = 1$ and $b_2 = 1$. Then we can write

$$E_S = SPR \times TE_S = \left(\left\lceil \frac{(S \times IE_{UCP} \times (1+Cost_S \times RV))}{62N+b_1(-1.5N-7D)+b_2(-1.5N-4A)} \right\rceil + 1\right) \times [80\,N + 18 + b1\,(-5.5\,N + 21\,D - 18) + b2\,(-1.5\,N + 7\,A)]$$

$$T_S = \left(\left\lceil \frac{(S\ \times IE_{UCP} \times (1 + Cost_S \times RV))}{62N + b_1(-1.5N - 7D) + b_2(-1.5N - 4A)} \right\rceil + 1\right) \times 80$$

2. *Estimating Effort per UCP*

Some research about estimating effort per UCP is available for both Waterfall and agile, e.g. Schneider and Winters (2001), Ani and Basri (2013), and Karner (1993). However, these estimations have been criticized by Ribu, and it is recommended that each organization estimate effort per UCP based on its own historical data (Ribu, 2001). If no historical data is available a value for effort per UCP can be assumed to be 20 man-hours/UCP (Karner, 1993). For Scrum, the implementation effort per UCP discussed in the model does not include requirements and high level design. Knowing that requirements and high level design is 24% of total project effort, we can assume that the implementation effort per UCP is effort per UCP×0.76 (Tan, 2012). Therefore, a typical value for IE_{UCP} is 15.2.

3. Accounting for Requirements Volatility

Requirement volatility, *RV,* is the percentage requirements that are expected to change across the project. According to Peña, the average volatility of a project is 22% with standard deviation 16% (Peña, 2012).

Cost of change is the cost it takes to implement a change. This cost varies depending on the percentage effort completed when this change arrives. Moreover, change doesn't arrive uniformly over the project lifecycle.

In case historical data is available regarding the cost of change across the project, this data should be used to find values for C_W and C_S , the cost of change for Waterfall and Scrum accordingly. However, in the absence of historical data, we find reasonable values for C_W and C_S as we will discuss.

Since we assumed that the project is always successful, we should assume that the requirements volatility will decrease with time with no volatility arriving at the end of the project. Otherwise, the Waterfall model might face difficulties that might lead to failure. Therefore, we assume that the requirements volatility distribution as a function of percent effort completed follows a triangular distribution as shown in Figure 8, where no volatility arrives during the last 10% of the project.

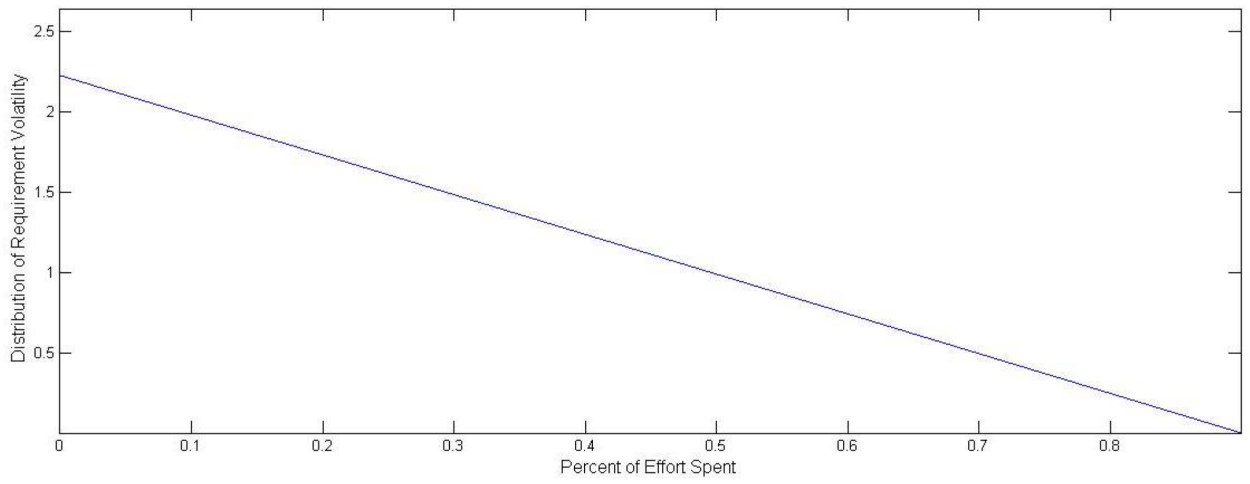


Figure 8: Requirement Volatility Distribution as a Function of Percent Effort Spent

For Waterfall the relative cost of change depends on the phase where the change arrives. For the requirements phase, the relative cost to fix is 1, for the design phase the relative cost to fix is 5, for the code phase the relative cost to fix is 10, for the testing phase, the relative cost to fix is 50, and for the operations phase the relative cost to fix is 100 or more (Boehm,1981).

In order to model the relative cost of change for Waterfall in a way that reflects this exponential increase with project progress, the Waterfall relative cost to change is assumed to be 100^x, where x is the percent effort completed as shown in Figure 9. According to this assumption, the relative cost to fix increases exponentially as the project progresses varying from 1 to 100.

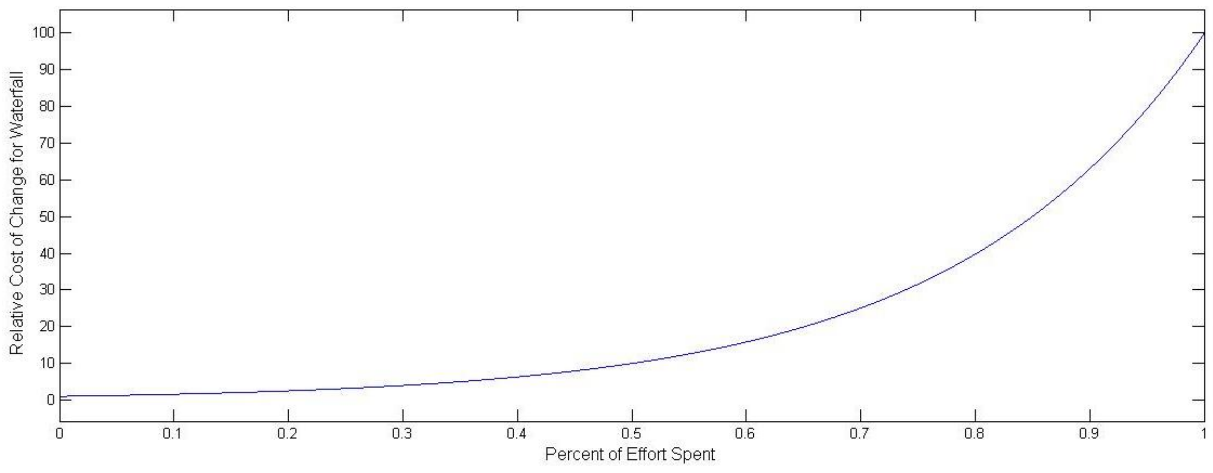


Figure 9: Waterfall Relative Cost of Change as a Function of Percent Effort Spent

For agile development, the cost of change curve can be assumed to be of the order *O(log(n))* (Cockburn, 2000). We model the relative cost of change for Scrum as *log(100x+1)* where x is the portion of project completed, and log is the natural logarithm as shown in Figure 10.

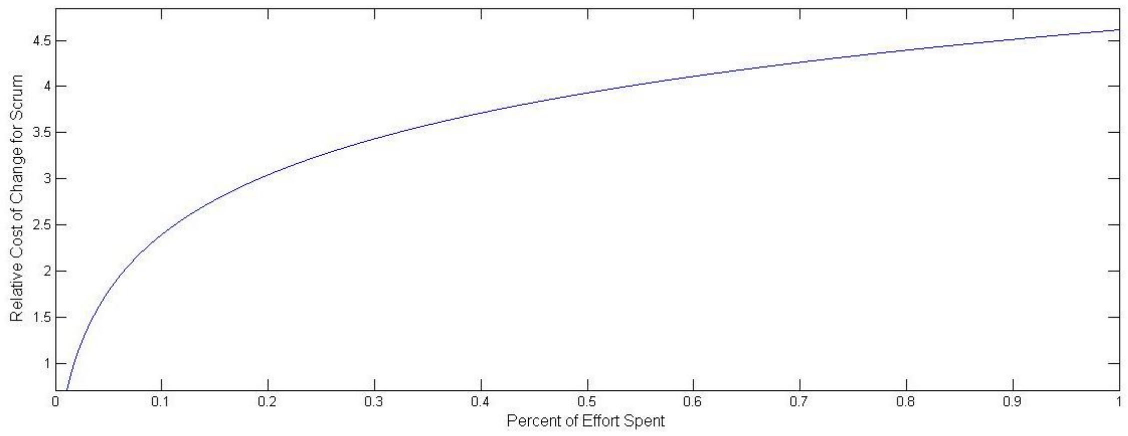


Figure 10: Scrum Relative Cost of Change as a Function of Percent Effort Spent

Therefore, for Waterfall the total cost of change over the course of the project is

$$Cost_W = \int_0^{0.9} \left(\frac{2\times(0.9-x)}{0.9^2} \times 100^x\right) dx = 6.747.$$

For Scrum, the total cost of change over the course of the project is

$$Cost_S = \int_0^{0.9} \left(\frac{2 \times (0.9 - x)}{0.9^2} \times \log(100x + 1)\right) dx = 3.1.$$

4. *The Final Model*

For Waterfall, the total effort is expressed as

$$E_W = S \times E_{UCP} \times (1 + RV \times Cost_W)$$

According to COCOMO, assuming the project has somewhat flexible constraints, the project time and number of developers required to finish the project can be expressed as (Merlo–Schett et. al, 2003),

$$T_W = 2.5 \times (E_W)^{0.35}$$

$$N = \frac{E_W}{T_W}$$

The total effort of the Scrum model is found by solving the following optimization problem, where *N* (team size) is the decision variable, and *D*, *A*, b_1, and $b_2$are auxiliary decision variables.

Minimize $E_S = SPR \times [80\,N + 18 + b1\,(-5.5\,N + 21\,D - 18) + b2\,(-1.5\,N + 7\,A)]$

Subject to

$$SPR = (\left\lceil \frac{S \times IE_S \times (1 + RV \times Cost_S)}{DE_S} \right\rceil + 1)$$

$D = \left\lceil \frac{N}{10} \right\rceil$

$A = \left\lceil \frac{D}{10} \right\rceil$

$b_1 \equiv D > 10$

$b_2 \equiv A > 10$

$DE_S = 62N + b_1(-1.5N - 7D) + b_2(-1.5N - 4A)$

$N > 2$

$N \text{ is an integer}, b_1 \text{ and } b_2 \text{ are binary}.$

We are concerned with the ratio

$$R = \frac{E_S}{E_W}$$

The Scrum development method minimizes effort, and is therefore preferred over Waterfall, if and only if $R < 1$.

CHAPTER V

ANALYSIS

A. Illustrative Example

To show how the model can be applied, we present an illustrative example. Consider a project of size 500 UCP that has an average requirement volatility of 22% over the project lifecycle. We assume that the effort needed per UCP is 20 man-hours, which is the value suggested by Karner (Karner, 1993). Consequently, we assume that the implementation effort per UCP is $E_{UCP}\times$ Ratio of implementation effort = 20×0.76=15.2 man-hours.

In this case, applying the Waterfall model to these given values results in a required effort of 155.3 person-months, 11 developers, and 14.6 months, while applying the Scrum optimization model results in a minimum effort of 110.7 person-months, an optimal number of developers 9 people and required time of 12 months. According to the suggested model, Scrum is better for minimizing effort in this case.

B. Waterfall Model

Applying the same parameters of the illustrative example to different project sizes and requirement volatility levels for the Waterfall model yields a range of required efforts presented in Figure 11. Average volatility refers to a requirement volatility of 22%, low volatility refers to a requirement volatility of 6% which is one standard deviation below the average (22-16), high volatility refers to a requirement volatility of 38% which is one standard deviation above the average (22+16), and no requirement volatility refers to a requirement volatility of 0%.

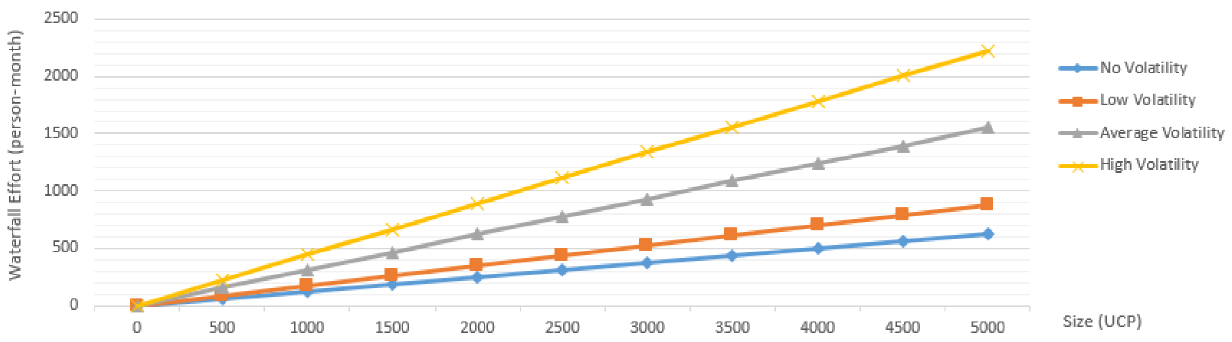


Figure 11: Waterfall Effort as a Function of Project Size

We note that for a given requirement volatility level, the required effort for Waterfall is linear as a function of project size. The higher the volatility level is, the steeper the slope. The required time and number of developers for different project sizes under various volatility conditions are shown in Figures 12 and 13 below. These functions are as a direct result of applying the formulas suggested by the COCOMO model.

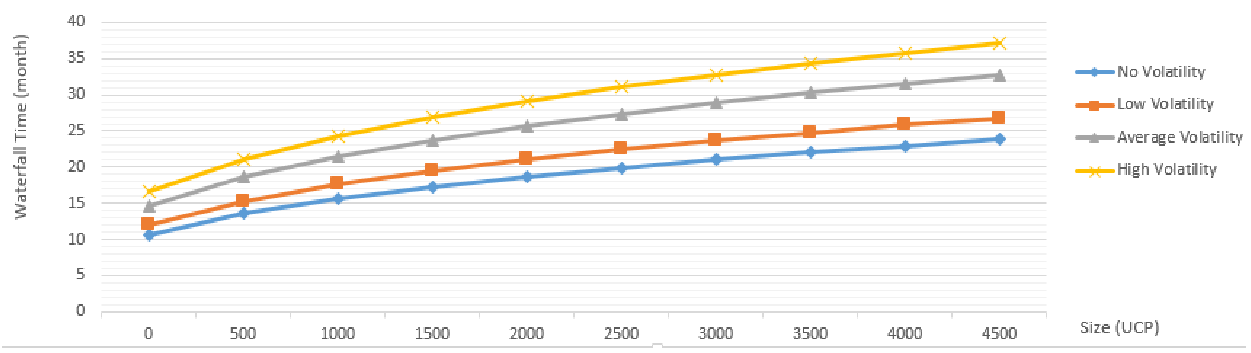


Figure 12: Waterfall Time as a Function of Project Size

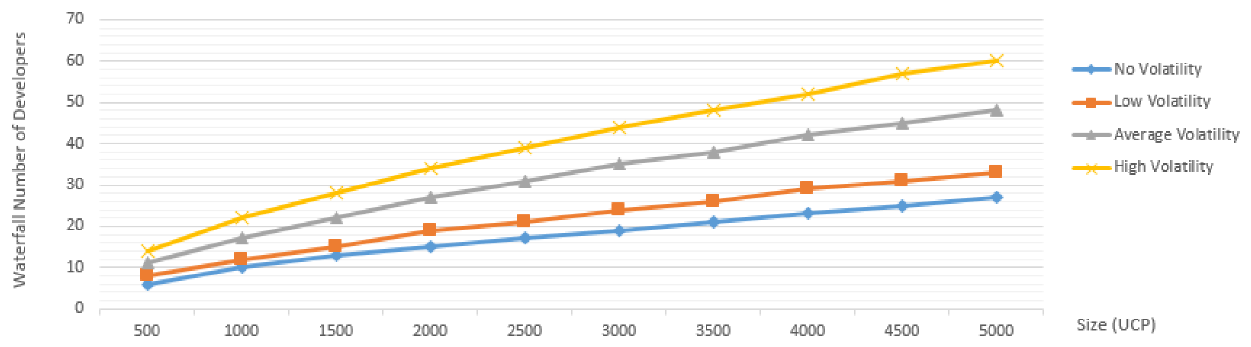


Figure 13: Waterfall Number of Developers as a Function of Project Size

C. Scrum Model

Applying the same parameters of the illustrative example to different project sizes and requirement volatility levels for the Waterfall model yields a range of required efforts presented in Figure 14. These efforts are derived by solving the optimization problem of minimizing effort with the number of developers as a decision variable. We notice that the Scrum model is less sensitive to requirements volatility and to project size. This means that Scrum is better at handling change and it scales up better than Waterfall. Since the Scrum methodology was modeled as a non-linear optimization model, a formalized solution is difficult to find.

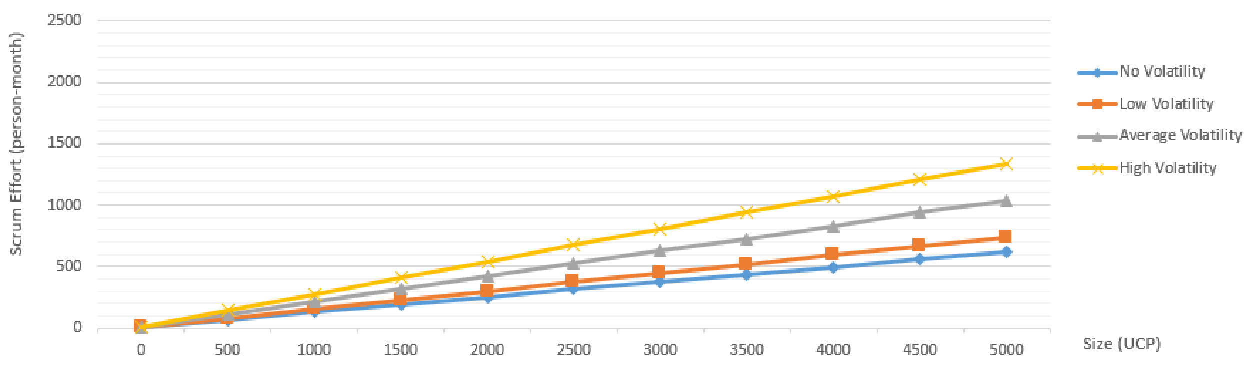


Figure 14: Scrum Effort as Function of Project Size

The optimal number of developers for different project sizes under various requirements volatility levels is shown in Figure 15. We notice from the results that unlike Waterfall, for high volatility levels, and for bigger projects, the optimal number of developers only slightly increases. We also notice that in some cases, a bigger project requires fewer developers in order to minimize effort. This is due to the nonlinear nature of the Scrum model suggested.

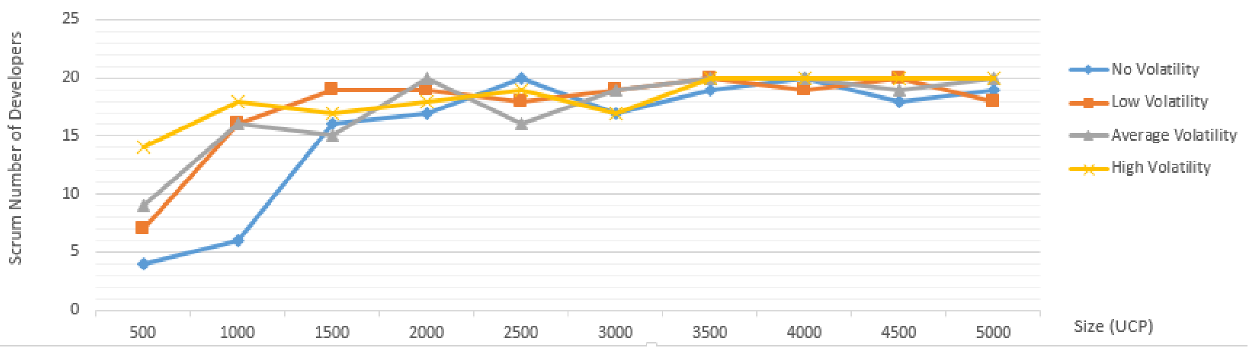


Figure 15: Scrum Number of Developers as a Function of Project Size

As for the time needed to complete a project following the Scrum methodology and minimizing effort, the results are shown in Figure 16. The results indicate that there are cases where a bigger project needs less time than a smaller one, or when a project with no requirements volatility needs more time than one with high volatility. However, the pattern in general shows that for bigger projects and higher volatility more time is needed. The reason for this variation is because the optimal number of developers as shown previously does not behave linearly with respect to requirement volatility or project size.

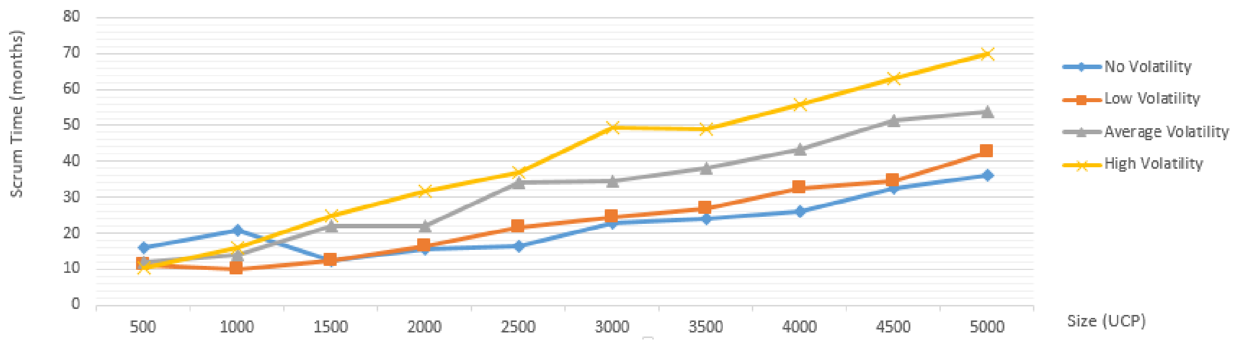


Figure 16: Scrum Time as a Function of Project Size

D. Waterfall versus Scrum

Although analyzing the Waterfall model and Scrum model individually is important, we are mostly concerned with knowing which model is more suitable under given circumstances. Particularly, we are interested in the ratio of Waterfall effort to Scrum effort. Plotting this ratio as a function of project size under different volatility rates yields the results shown in Figure 17. According to the results, Scrum is favorable under most conditions. Waterfall is only favorable under conditions where there is no or very low requirements volatility. Moreover, for bigger projects, Scrum is preferred over Waterfall.

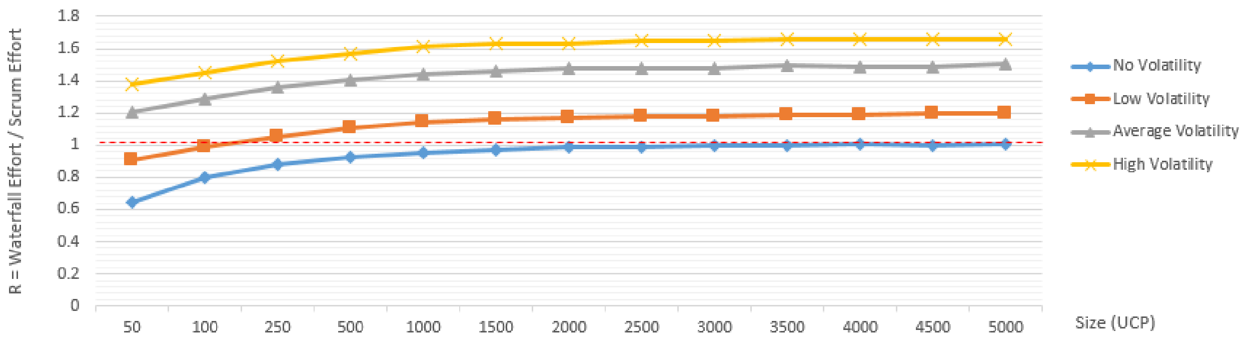


Figure 17: Ratio of Waterfall Effort to Scrum Effort versus Project Size

As for the number of developers needed to complete the project, the ratio of number of developers needed when following Waterfall according to COCOMO to the number of developers needed following the Scrum model that optimizes effort as a function of project size is shown in Figure 18. For large projects, Waterfall requires a large number of people to finish the project, especially in the case of high requirements volatility. However, when the project size is not large, it is not always clear which method requires more developers.

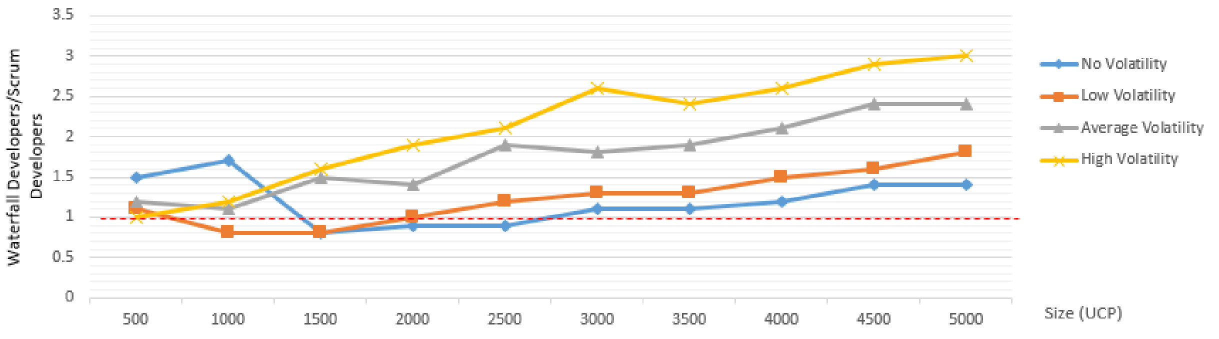


Figure 18: Ratio of Waterfall Number of Developers to Scrum Number of Developers as a Function of Project Size

Consequently, it is not strange that for large projects Scrum projects require more time that Waterfall projects as shown in Figure 19. Since the objective of the model was to minimize effort, having a smaller number of developers results in needing more time to finish the project. In case there is a project deadline to meet, a time constraint could be added to the optimization model. This would lead to a different set of solutions where Waterfall might prove to be better than Scrum in some circumstances where it had been considered to be worse without the time constraint. Moreover, in case the customer is interested in minimizing the time needed to finish the project, the model should be altered to have the objective of minimizing time with the number of developers as a decision variable.

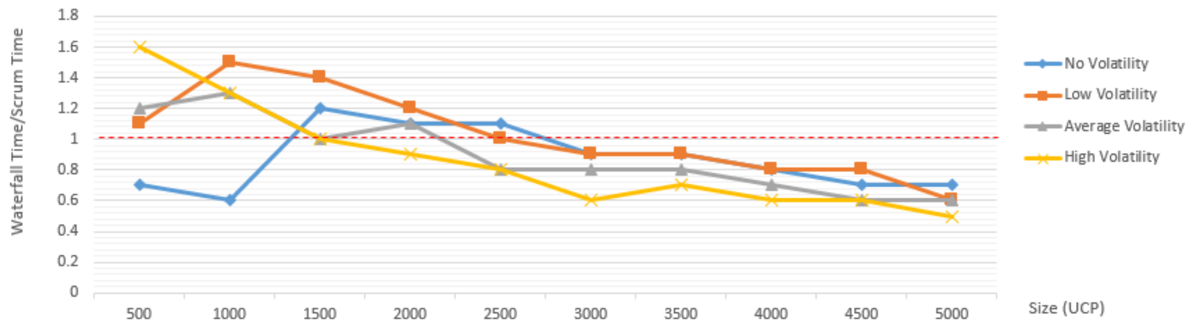
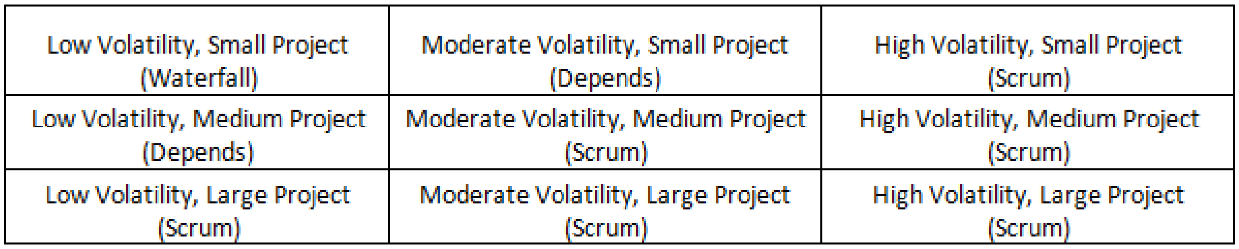


Figure 19: Ratio of Waterfall Time to Scrum Time as a Function of Project Size

We summarize the basic findings on the choice of software development method depending on size and volatility in the framework shown in Figure 20. These results are applicable under the values we considered in the illustrative example. For different values for effort per UCP, implementation effort per UCP, cost of change for Waterfall and Scrum, and requirement volatility levels, this framework might not apply, although the general idea that Scrum is better under high volatility and large projects might still be viable.



Low Volatility, Small Project (Waterfall)	Moderate Volatility, Small Project (Depends)	High Volatility, Small Project (Scrum)
Low Volatility, Medium Project (Depends)	Moderate Volatility, Medium Project (Scrum)	High Volatility, Medium Project (Scrum)
Low Volatility, Large Project (Scrum)	Moderate Volatility, Large Project (Scrum)	High Volatility, Large Project (Scrum)

Figure 20: Choice of Model Depending on Size and Volatility

E. Comparing the Results to the Literature

Our model supports the literature suggesting that agile is more suitable for highly volatile projects while plan-based methods are more appropriate for conditions under which the requirements are stable. Moreover, as suggested by the literature, our model also suggests that for bigger project size, the number of developers needed by agile is less than that needed by plan-based methods.

However, while most of the comparative literature suggests that agile is more suited for smaller projects and traditional methods work better for big projects, our model suggests the opposite. Perhaps one reason for that discrepancy is that Scrum of Scrums might be often ignored or underestimated when comparing agile to plan-based methods.

CHAPTER VI

CONCLUSION AND FUTURE WORK

All in all, we have presented a comparative model that aims to minimize the effort needed to complete a project by choosing the optimal software development method. Our proposed model takes into account the different structures of the Waterfall of Scrum software development methods, project size, implementation effort per use case point (productivity), and the effects of requirements volatility with the team size as a decision variable. We proved quantitatively that Scrum is more suitable for situations where there is high requirement volatility. Moreover, we showed that Scrum is better for large projects, while Waterfall is better for smaller projects.

Our results agree with the literature suggesting that agile methods are better for highly volatile environments, but they do not concur with the literature suggesting that traditional methods are better than agile methods for large projects. A deeper investigation of how large projects behave when following Scrum of Scrums can shed light on the flexibility and scalability of Scrum.

One limitation of this study is that inputs to the model such as implementation effort per UCP and cost to change for Waterfall and Scrum were suggested based on loose assumptions. These values depend on historical data for accuracy. Another limitation is that the success of the project is taken for granted. A major concern in software engineering is maximizing project success in terms of achieving all the required features within the required time and budget. In this aspect, Waterfall and Scrum are incomparable. Moreover, we consider that enough resources are allocated to address any requirement volatility at any time. In reality, there might be a certain percentage of effort that is allocated for customer involvement during a given phase in the software development lifecycle. If the available resources in one phase are not enough to deal with customer requests to change, these changes need to be carried over to future phase where cost of change is higher.

In order to address these limitations, future work could include improving on our model by getting more accurate estimates to some of the model inputs such as implementation effort per UCP and cost to change for Waterfall and Scrum. In addition, other models such as one that compares Spiral development to Scrum can be developed with the objective of maximizing project success taking into account resources allocated for customer involvement. Such a model would address concerns related to success or failure of the project in light of requirements volatility and the ability of these models to handle volatility that could arrive late in the project lifecycle.

REFERENCES

Adolph, S., Cockburn, A., & Bramble, P. (2002). *Patterns for effective use cases*. Addison-Wesley Longman Publishing Co., Inc.

Ahmed, A., Ahmad, S., Ehsan, N., Mirza, E., & Sarwar, S. Z. (2010, June). Agile software development: Impact on productivity and quality. In *Management of Innovation and Technology (ICMIT), 2010 IEEE International Conference on* (pp. 287-291). IEEE.

Alliance, A. (2001). Agile manifesto. *Online at* http://www.agilemanifesto.org.

Anda, B., Dreiem, H., Sjøberg, D. I., & Jørgensen, M. (2001). Estimating software development effort based on use cases—experiences from industry. In *≪UML≫ 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools* (pp. 487-502). Springer Berlin Heidelberg.

Ani, Z. C., & Basri, S. (2013). A CASE STUDY OF EFFORT ESTIMATION IN AGILE SOFTWARE DEVELOPMENT USING USE CASE POINTS. *Science International*, *25*(4).

APLN, V. O. (2013). 2nd Annual Survey. The State of Agile. *VersionOne website*.

Awad, M. A. (2005). A comparison between agile and traditional software development methodologies. *University of Western Australia*.

Bajwa, S. S. (2009). Investigating the Nature of Relationship between Software Size and Development Effort. *Department of Interaction and System Design, Blekinge Institute of Technology. Master of Science*.

Beck, K. (1999). Embracing change with extreme programming. *Computer*,*32*(10), 70-77.

Beck, K. (2000). *Extreme programming explained: embrace change*. Addison-Wesley Professional.

Boehm, B. (2006). A view of 20th and 21st century software engineering. In *Proceedings of the 28th international conference on Software engineering* (pp. 12-29). ACM.

Boehm, B. W. (1981). *Software engineering economics* (Vol. 197). Englewood Cliffs (NJ): Prentice-hall.

Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, *21*(5), 61-72.

Boehm, B. W. (2007). Software engineering economics. *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*, *69*, 117.

Boehm, B. W., Brown, J. R., & Kaspar, H. (1978). Characteristics of software quality.

Boehm, B., & Turner, R. (2003, June). Observations on balancing discipline and agility. In *Agile Development Conference, 2003. ADC 2003. Proceedings of the* (pp. 32-39). IEEE.

Boehm, B., Abts, C., Clark, B., & Devnani-Chulani, S. (1997). COCOMO II model definition manual. *The University of Southern California*.

Boehm, B., Valerdi, R., & Honour, E. (2008). The ROI of systems engineering: Some quantitative results for software intensive systems. Systems Engineering, 11(3), 221-234.

Bundschuh, M., & Dekkers, C. (2008). The IFPUG Function Point Counting Method. *The IT Measurement Compendium: Estimating and Benchmarking Success with Functional Size Measurement*, 323-363.

Cao, L. (2005). *Modeling dynamics in agile software development*. Georgia State University.

CapTech. (2015). A Comparative Look at Top Agile Tools. *Online at https://www.captechconsulting.com/blogs/a-comparative-look-at-top-agile-tools*

Cheung, Y., Willis, R., & Milne, B. (1999). Software benchmarks using function point analysis. *Benchmarking: An International Journal*, *6*(3), 269-276.

Cocco, L., Mannaro, K., Concas, G., & Marchesi, M. (2011). Simulating kanban and scrum vs. waterfall with system dynamics. In *Agile Processes in Software Engineering and Extreme Programming* (pp. 117-131). Springer Berlin Heidelberg.

Cockburn, A. (2000). Reexamining the cost of change curve. *on-line: http://ronjeffries.com/xprog/articles/cost_of_change/*.

Cohn, M. (2004). *User stories applied: For agile software development*. Addison-Wesley Professional.

Cohn, M., & Ford, D. (2003). Introducing an agile process to an organization.*Computer*, *36*(6), 74-78.

Dubakov, M., & Stevens, P. (2008). Agile Tools: The Good, the Bad and the Ugly. *Report, TargetProcess, Inc.*

Easterbrook, S. (2001). Software Lifecycles. *University of Toronto Department of Computer Science*.

Eeles, P. & Houston, K. (2002). Building J2EE applications with the rational unified process. Addison-Wesley Longman Publishing Co., Inc..

Felhman, T., & Santillo, L. (2010). From Story Points to COSMIC Function Points in Agile Software Development–A Six Sigma perspective. In *International Workshop on Software Measurement–IWSM*.

Goyal, S (2008). Agile Techniques for Project Management and Software Engineering.

Harter, D. E., Krishnan, M. S., & Slaughter, S. A. (2000). Effects of process maturity on quality, cycle time, and effort in software product development.*Management Science*, *46*(4), 451-466.

Heidelberg. June, 2014.

Hoda, R., Noble, J., & Marshall, S. (2010). Agile undercover: when customers don't collaborate. In *Agile Processes in Software Engineering and Extreme Programming* (pp. 73-87). Springer Berlin

Horowitz, E. (1994). USC COCOMO Reference Manual. In *University of Southern California.*

Huskins, M. Kaplan, J.,& Krishnakanthan, K. (2013). Enhancing the efficiency and effectiveness of application development. McKinsey & Company.

Jansi, S., & Rajeswari, M. K. (2015). A Greedy Heuristic Approach for Sprint Planning in Agile Software Development.

Jones, C. (2007). Estimating software costs. *McGraw-Hill*

Kan, S. H. (2002). *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc.

Karner, G. (1993). Resource estimation for objectory projects. *Objective Systems SF AB*, *17*.

Koushik, M., Mookerjee, V. (1995). Modeling Coordination in Software Construction: An analytical approach

Kulk, G. P., & Verhoef, C. (2008). Quantifying requirements volatility effects.*Science of Computer Programming*, *72*(3), 136-175.

Kumar, G., & Bhatia, P. K. (2012). Impact of Agile Methodology on Software Development Process. *International Journal of Computer Technology and Electronics Engineering (IJCTEE) Volume*, *2*.

Larman, C., & Vodde, B. (2013). Scaling Agile Development. *CrossTalk*, 9.

Lindvall, M., Basili, V., Boehm, B., Costa, P., Dangle, K., Shull, F., & Zelkowitz, M. (2002). Empirical findings in agile methods. In *Extreme Programming and Agile Methods—XP/Agile Universe 2002* (pp. 197-207). Springer Berlin Heidelberg.

Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., & Kahkonen, T. (2004). Agile software development in large organizations.*Computer*, *37*(12), 26-34.
McCall, J. A., Richards, P. K., & Walters, G. F. (1977). *Factors in software quality*. General Electric, National Technical Information Service.

Merlo–Schett, N., Glinz, M., & Mukhija, A. Seminar on Software Cost Estimation WS 2002/2003.

Mohammed, N., Munassar, A., & Govardhan, A. (2010). A Comparison Between Five Models Of Software Engineering.

Moniruzzaman, A. B. M., & Hossain, D. S. A. (2013). Comparative Study on Agile software development methodologies. *arXiv preprint arXiv:1307.3356*.

O'Regan, G. (2008). *A brief history of computing*. Springer.

Peña, M. E. (2012). *Quantifying the impact of requirements volatility on systems engineering effort*. University of Southern California.

Ribu, K. (2001). Estimating object-oriented software projects with use cases. In *University of Oslo, Dept*.

Rowe, G., & Wright, G. (1999). The Delphi technique as a forecasting tool: issues and analysis. *International journal of forecasting*, *15*(4), 353-375

Rowen, R. B. (1990). Software project management under incomplete and ambiguous specifications. *Engineering Management, IEEE Transactions on*,*37*(1), 10-21.

Royce, W.W. “Managing the Development of Large Software Systems,”1-9. Proceedings of IEEE WESCON, August 1970. IEEE, 1970 (originally published by TRW).

Schneider, G., & Winters, J. P. (2001). *Applying use cases: a practical guide*. Pearson Education.

Shao, B., Yin, P. Y., & Chen, A. N. (2014). Organizing knowledge workforce for specified iterative software development tasks. *Decision Support Systems*, *59*, 15-27.

Shepperd, M., Schofield, C., & Kitchenham, B. (1996, May). Effort estimation using analogy. In *Proceedings of the 18th international conference on Software engineering* (pp. 170-178). IEEE Computer Society.

STANDISH GROUP. (2013). The CHAOS Manifesto–Think Big, Act Small, last accessed on 27 June, 2014.

Sutherland, J. (2010). Scrum handbook. *Online (12.10. 2012): jeffsutherland. com/Scrumhandbook. pdf*.

Sutherland, J., & Schaber, K. (2013). The Scrum Guide TM.

Tan, T. (2012). *Domain-based effort distribution model for software cost estimation* (Doctoral dissertation, University of Southern California).

Thakurta, R., & Ahlemann, F. (2010, January). Understanding requirements volatility in software projects-an empirical investigation of volatility awareness, management approaches and their applicability. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on* (pp. 1-10). IEEE.

West, D., Grant, T., Gerush, M., & D’silva, D. (2010). Agile development: Mainstream adoption has changed agility. *Forrester Research*, *2*, 41.

Yang, Y., He, M., Li, M., Wang, Q., & Boehm, B. (2008, October). Phase distribution of software development effort. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement* (pp. 61-69). ACM.

APPENDIX

For completion, we list the agile manifesto principles to give a clearer idea about agile, and we list other models to consider for future work such as the Spiral Model, the Rational Unified Process, Extreme Programming, and Feature Driven Development.

I. Agile Manifesto Principles

“Our highest priority is to satisfy the customer through early and continuous delivery
of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity--the art of maximizing the amount of work not done--is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly"

II. Software Development Methods

Spiral Model

The Spiral Model (Boehm, 1988) is an iterative model that focuses on reducing risk. This model has four sectors which are repeated throughout the project lifecycle in an iterative manner where each iteration is called a spiral. These sectors are: Objective setting where specific objectives for the phases are identified, risk assessment and reduction where the activities are prioritized to reduce risk though risk analysis and prototyping, development and validation, and planning of the next phase.

As shown in Figure 21 (Boehm, 1988), each spiral in the model represents a phase or a round of software development. There is a specific round for each of feasibility study, concept of operation, top level requirements and specifications, software design, and implementation of the system. Each of these rounds in turn passes through the four sectors of the spiral model.

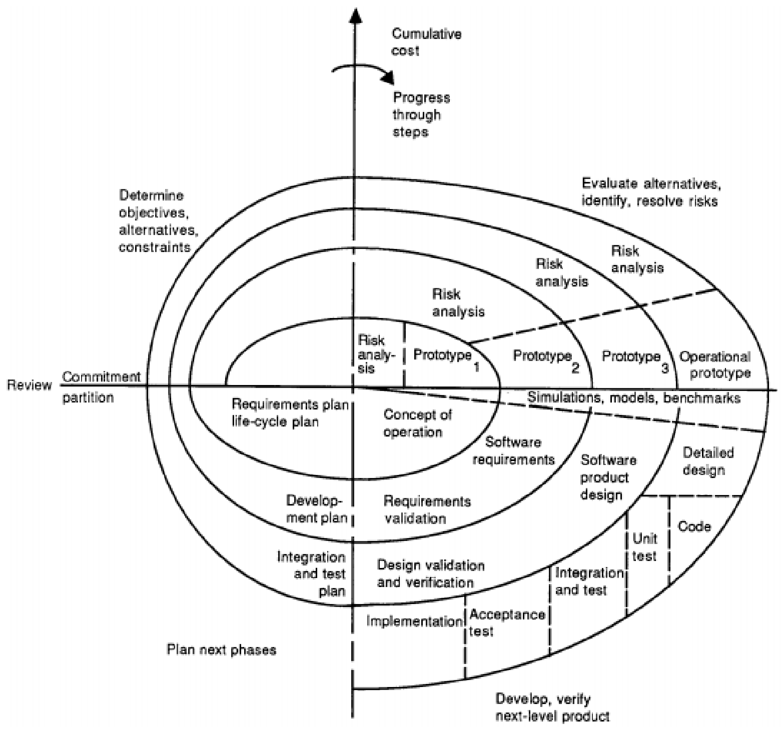


Figure 21: The Spiral Model for Software Development (Boehm, 1988)

Rational Unified Process

Another popular iterative model is the Rational Unified Process (RUP). The Rational Unified Process divides software development into four phases: Inception, elaboration, construction, and transition. It also identifies core processes and how they are distributed along the phases of the model as shown in Figure 22 (Eeles and Houston, 2002). Moreover, the phases of the software development lifecycle are further divided into iterations where the first few iterations focus on activities such as business modeling and requirements (inception processes), while the last few iterations focus on testing and deployment activities (transition processes).

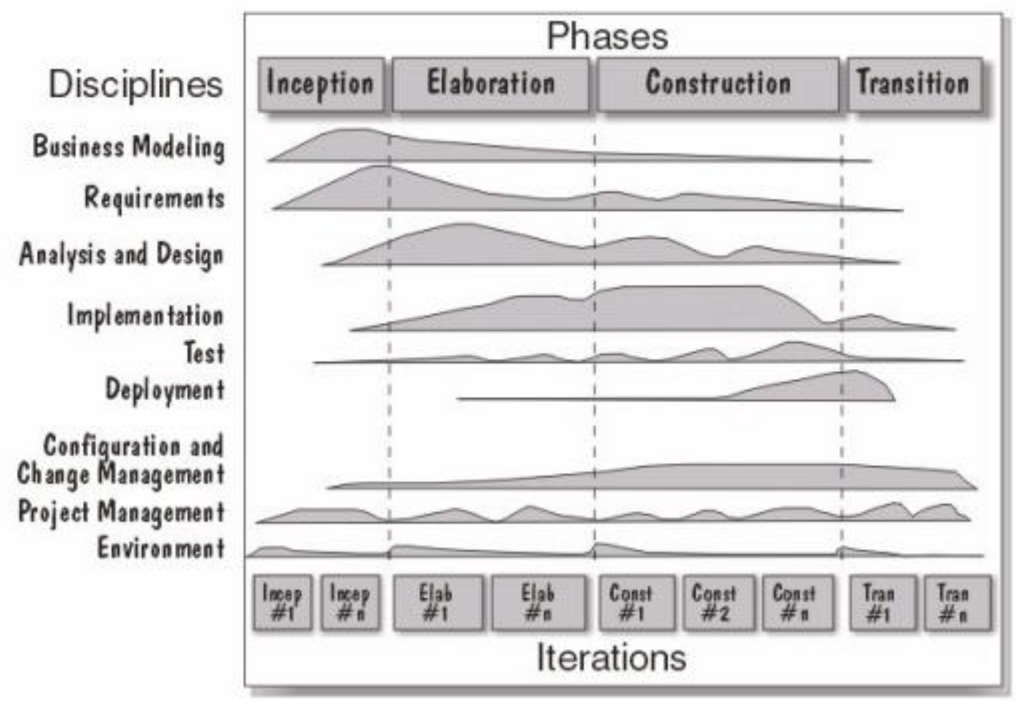


Figure 22: The Rational Unified Process of Software Development (Eeles and Houston, 2002)

Extreme Programming

Extreme Programming is a well-known agile method that was created in the 1990's and is still used by many software developers nowadays. The driving values of the Extreme Programming (XP) method are communication, simplicity, feedback, courage, and respect.

Based on these values, principles and practices that achieve flexible and effective development are identified. The practices implemented by Extreme Programming supporters include but are not limited to pair programming, incremental development, customer involvement, shared code ownership, planning for releases, continuous integration, and test-first programming. While some of these practices can be implemented in other software development methods, they work well together under the umbrella of extreme programming and have shown success in the software industry (Beck, 1999).

Extreme Programming follows an iterative development process. Requirements are identified for each release or portion of the product. During the implementation of each release, several iterations take place to further divide the work into smaller tasks and accommodate changes. Before each release and iteration, a release planning meeting and an iteration planning meeting are held accordingly. Feedback from acceptance tests guides the development of future iterations and releases. Figure 23 (Beck, 2000) describes the Extreme Programming method workflow.

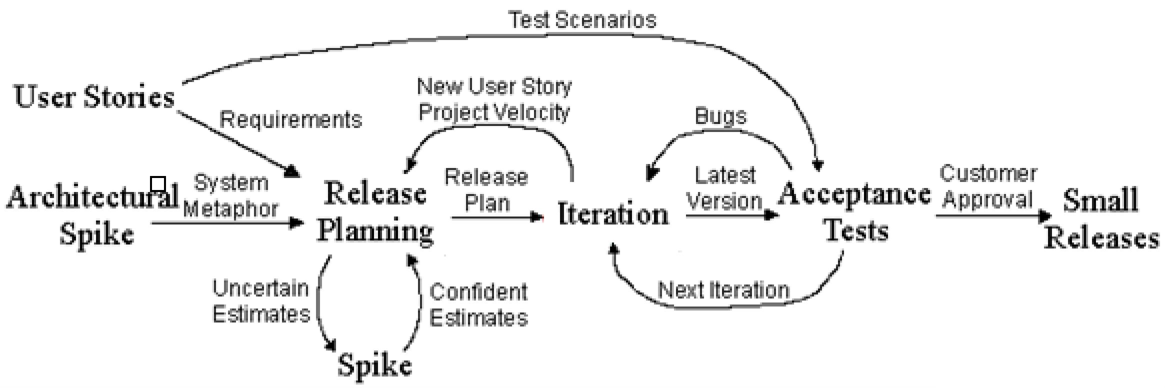


Figure 23: Workflow of the Extreme Programming Method [adopted from Beck, 2000]

Feature Driven Development

As the name implies, the Feature Driven Development method divides the project into features. A feature is defined as a function that maps to a step in some activity and that can be completed in less than two weeks. This method consists of five main processes:

Developing an overall model, building a features list, planning by feature, designing by feature, and building by feature.

This method assumes that an object oriented (OO) approach is used in software development and gives special attention to OO concepts such as relationships between objects and classes. The Feature-Driven Development approach assigns a responsible individual for every class (unit of code in OO). Teams that consist of different class owners work together to design and build a feature in an iterative manner. Since there is regular building by feature, a demo is always readily available for customers (Goyal, 2008).