A Comparison of Software Project Architectures: Agile, Waterfall, Spiral, and Set-Based

by

Christian J West

B.Sc. Computer Science
Utah State University, 2007

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

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

JUNE 2018

Signature redacted

Signature of the Author: .....
Graduate Fellow, System Design and Management Program
May 11, 2018

Signature redacted

Certified by: .....
Bryan R Moser
Academic Director and Senior Lecturer, System Design & Management Program
Thesis Supervisor

Signature redacted

Accepted by: .....
Joan Rubin
Executive Director, System Design & Management Program

This page is intentionally left blank

A Comparison of Software Project Architectures: Agile, Waterfall, Spiral, and Set-Based

by

Christian J West

Submitted to the System Design and Management Program
on May 2018 in Partial Fulfillment of the
Requirements for the Degree of Master of Science in
Engineering and Management

ABSTRACT

Engineers and managers of software projects have the daunting task of successfully delivering the right product at the right time for the right price. The path to achieving these lofty goals is commonly not a straightforward endeavor. Due to the dynamic nature of software development, varying organizational circumstances, and situational idiosyncrasies of each project this can be a very difficult and sensitive process. Ideally, software development methodologies bring order to the chaos of software development. An ill-fitting development strategy, however, can create unnecessary friction and further complicate the prospect of a successful product delivery. Researchers and private organizations alike spend substantial resources to understand the strengths and weaknesses of commonly used development practices — the validation of which is highly problematic due to conflicting variables. This research ventures to bring clarity to the question: "Which development methodology is right for a particular situation?" Treating the software development project life-cycle as a socio-technical system, it can be decomposed to the most fundamental elements. Using these elements as the architectural building blocks of a project, Agile, Waterfall, Set-Based, and Spiral are each compared at the molecular level. This thesis evaluates these comparisons and how subsequent research applies to today's software projects.

Thesis Supervisor: Bryan R Moser
Academic Director and Senior Lecturer, System Design & Management Program

This page is intentionally left blank

Acknowledgements

My time at MIT has been special for me and my family. It's pushed each of us beyond our own perceived limits in every sense of the word. With four children, this has been a family affair for my wife and I from day one. The thesis process has been a culmination of our MIT experience. Researching and writing the thesis has contained many of the university's hallmarks: high expectations, intense pressure, and late nights. But more importantly: questions, curiosity, and discovery. Many contributed to making this a reality for me. I'm grateful for a chance to recognize them.

First, my classmates who encouraged me to think beyond SDM: Adrien Moreau, José Garza, Ben Linville-Engler, Justin Burke, and Tim Chiang. Travis Rapp, for the discussions on Set-Based design. Paula Ingabire, who understands the difficulties of writing a thesis at a distance while working full-time and inspired me to keep writing. Max Reele, for the early morning debate sessions, steady encouragement, and thoughtful insight. I'd like to thank Professor Warren Seering for his interest and willingness to share his knowledge. The SDM staff who work tirelessly to support their students and have made this experience so special: Joan Rubin, Bill Foley, Amanda Rosas, Naomi Gutierrez, Jon Pratt, and so many others. To my parents, Richard and Sharon West, and my wife's parents, Steve and Dona Reeder, who in so many ways have made this journey possible.

I'd like to thank Professor Bryan Moser for his genuine concern for me and his passion for research. His counsel and direction have been invaluable to me and have made not only this thesis successful, but also my entire MIT experience. To my four wonderful children: CJ, Soren, Jane, and Nellie. Who have come to think of thesis as a naughty word and who never cease to provide ample distractions. And to my wife, Sadie. She has sacrificed everything to allow me to take this journey. She has struggled alongside me from the beginning to the very end. In regard to this thesis, she prioritized employing her superhuman editing skills for my benefit. I love you and am so grateful to be in the trenches of life with you.

This page is intentionally left blank

Table of Contents

List of Figures

List of Tables

List of Abbreviations

XP ..... Extreme Programming

RAD ..... Rapid Application Development

FDD ..... Feature Driven Development

TDD ..... Test Driven Development

SWEBOK ..... Software Engineering Body of Knowledge

KA ..... Knowledge Attributes (from SWEBOK)

SBCE ..... Set-Based Concurrent Engineering

DoD ..... Department of Defense

This page is intentionally left blank

1. Introduction

1.1. Motivation

I started out my career as a software engineer and technical co-founder of the software company, Meosphere. For 18 months, our small team of three to six engineers had only one product under management. Our user experience designer was the same person as our interface designer, and the president of the company sat two desks away from him. Our entire team occupied a few cubicles in the small co-working business complex where we rented our space. With such a small team and, relatively speaking, simple product, process was not a major concern. There weren't enough moving parts for us to benefit from a communication strategy, risk mitigation plans, or intricate review processes. Everything happening in the company was known by one or two key individuals. The simplicity of our situation allowed us to predict the impact of the decisions we made.

Three years later and in my second company, Lightning Kite, 25 engineers worked across five teams and 15-20 projects at varying stages in their life-cycle. We were building software systems for companies like Intel, Merck, the US Department of Health, and Blizzard. As opposed to my experience at Meosphere, where two or three engineers could understand all facets of the system in its entirety, we now needed processes in place in order to keep the dynamic projects from spinning out of control.

It was in these circumstances that we made the move from a waterfall software development strategy to a version of Agile called Scrum. I noticed good things resulting from the move. I felt the teams were more responsive to change, for example. However, there were also a number of areas where I thought we had regressed. But, I was never sure. I had no way to quantify the impact of the processes we used. Were we actually more productive, more efficient and ultimately more effective under Agile than Waterfall? I just didn't know.

I've since worked and interacted with teams and organizations of varying size and discussed my frustrations with other engineers, managers, and software business owners. These

discussions have convinced me that others share my concerns. As organizations and systems grow in size and number of components, they grow in complexity. An increase in complexity has direct impact on promises, delivery schedules and eventual market success of a product [1]. But with the right software management strategy, many of the negative effects of complexity can be mitigated. Alternatively, the application of ill-fitting strategies adds unnecessary friction for developing software and can lead to delayed, over-budget, and irrelevant products.

Professor Olivier de Weck, speaking at MIT in 2016, argued that as the complexity of a system increases, the time the project takes to complete and the error rates experienced also increase. This is due to the fact that the effort to complete a project grows at a super-linear rate compared to the increase in complexity of the project [1].

Managing projects at higher scale (i.e. higher complexity) presents larger risks to organizations, including missing target deadlines and overrun budgets. This drives us to find ways to develop methodologies for approaching projects that help manage increases in complexity and improve consistent delivery of value.

Since the early 2000's, development communities have been making a big shift and giving up methodologies such as Waterfall in favor of more agile and lean approaches. The result is much greater flexibility for software teams. However, as with most, if not all, software decisions, there are tradeoffs. In exchange for better adaptability to changing requirements or fuzzy direction, teams may be giving up accountability and vision. Is this tradeoff worth it? Is it successful at reducing project complexity? In some cases the answer is yes. However, in most cases the answer is a nuanced, *we think so*, or an even more genuine, *we really don't know*.

All these questions bring me to the motivation for this thesis, which is to help the companies building software products understand the impact of the methodologies they select.

1.2. Purpose

My purpose in this research was twofold: first, to identify which methodologies, or combination of methodologies, are the best fit for certain software project circumstances and their situational characteristics; and second, to identify emerging methodologies that are not now widely known.

While projects may be similar to one another in purpose or design, each one is unique in certain details. Considering these idiosyncrasies is crucial in the process of matching a design methodology to the project's structure and purpose, and even the setting of the project. Project variables may include strength of leadership, capabilities of the team, personnel turnover, changing project requirements, etc. Even from project to project within the same organization, the same team could obtain different outputs. The interactions between input and output variables are many and varied, and can be difficult to measure. Thus, it is very difficult for companies, managers, and engineers to say, "We've improved because we moved to X framework or methodology."

Notwithstanding those challenges, I set a course to provide evidence to those claims that a given approach will be more successful in a given application, so that a project manager may say with confidence, "We performed better [or worse] using Y methodology, and I can see that here."

For my second purpose, I was interested in conducting more detailed research on various methodologies. Perhaps there *is* an answer to the question: "Which methodology is best for this project?" Very likely, however, the answer could lie somewhere in between the canonical methodologies accepted today. Must the answer really be Agile *or* Waterfall *or* Scrum *or* Kanban *or* Spiral? Through my research, I would like to look at combinations of architectural decisions that make sound sense for teams to use — given their impact on project ilities.

1.3. Problem Statement

I've used the following problem statements as guides for developing the research. These hinge on the two components of the purpose of my research.

To	understand how different project management methodologies impact execution of software projects
By	comparing combinations of architectural decisions based on canonical methodologies
Using	a model to map outcomes across tradespaces of project ilities

To	discover new methodologies that have not yet been recorded
By	comparing combinations of architectural decisions outside of the canonical methods
Using	a model to map outcomes across tradespaces of project ilities

1.4. Hypothesis

I believe that quite often project managers and organizational leaders don't know if what they're doing is effective or not. There has been a substantial migration from Waterfall to Agile, which may be driven more by buzz in the industry rather than better project performance. By using an architectural decomposition, I hypothesized that project dynamics could be explained more completely and that methodologies could be more effectively matched and evaluated.

As I conducted this research, I expected to see some of the methodologies align with the traits they're known for (i.e. Agile for adaptability, Waterfall for looking ahead), but I also expected there to be some surprises. I expected to identify cases where people in the industry accepted a methodology for its specific benefits, but when the model simulations were completed, the results would indicate otherwise.

I expected to see Agile as particularly effective in projects with lower complexities, whereas Waterfall would be more effective in projects of greater size. In addition to Agile and Waterfall, I included Set-Based design and Spiral methodologies for comparison. I chose Spiral because it seems to be more rigid than Agile and more iterative than Waterfall, which makes it similar to two of the most popular methodologies but different enough to be worth considering. A set-based design, however, is like neither, but has qualities that make it interesting, such as developing multiple designs in parallel in order to arrive at an optimal solution. I was curious to see how it would compare to the others in practice.

And finally, I expected to see at least one new methodology emerge that could to be worth investigating further.

In this thesis, I first discuss the ways other researchers have ventured to evaluate software development methodologies and how effective they've been. I'll also look at mechanisms used to compare development strategies, followed by a brief overview of the main four methodologies considered in this thesis: Agile, Waterfall, Spiral, and Set-Based. I then discuss my research strategy of unpacking the architecture of software projects and use this to build a model to describe, categorize, and evaluate each methodology. I'll conclude by sharing the insights and learnings I've found through the research.

2. Literature Review

2.1. Evaluating Development Methodologies

Many approaches have been taken to evaluate the quality and effectiveness of software development methodologies. Some of these look at the software product output; some look at the byproducts of software development, such as lines of code or pull requests.

In [2], Barbara Kitchenham et al. cleverly stated in their review of software engineering methods and tools:

> *Software engineers split into two rather separate communities: those that build and maintain software systems and those who devise methods and tools that they would like the former group to use to build their software systems. To the disappointment of the tool and method developers, adoption of new methods and tool by builders of software is slow. To the disappointment of the builders, new methods and tools continue to proliferate without supporting evidence as to their benefits over existing approaches.*

New tools and methodologies seem to pop up each day. Wikipedia lists over 50 different methodologies, paradigms, processes and strategies [28]. Options include: Lean methodology, Kanban, Domain-driven design, Behavior-driven development, Rapid application development, and Test-driven development, to name only a few. With so many options to choose from, an evaluation strategy is an important way to determine if a certain methodology is right for a situation. In their paper on evaluating methodologies and proposing the DESMET method of evaluation, the authors of [2] suggest a list of nine possible approaches for evaluating software development strategies:

1) Quantitative experiment
2) Quantitative case study
3) Quantitative survey
4) Qualitative screening
5) Qualitative experiment

6) Qualitative case study
7) Qualitative survey
8) Hybrid method 1: Qualitative effects analysis
9) Hybrid method 2: Benchmarking

In addition to these methods, [2] also provides the conditions that favor each evaluation approach. For the purposes of the research for this thesis, the approach that most fit the conditions available is the Qualitative case study, or qualitative feature analysis. While the DESMET method no longer appears to be in use, the classifications the researchers created for evaluating methodologies proved useful for this research as they described the conditions required for each evaluation approach. This simplified the selection process of which approach to use for conducting this research.

When evaluating a methodology, it's very important to also consider the situational inputs where the project could be used. For example, the research by Inada in 2010 [3] describes the unique characteristics of software engineers and types of software built by Japanese software companies. The engineers found in this region are both educationally and culturally accustomed to following rules or working under strict discipline. When contrasting this set of situational inputs from that of a San Francisco-based mobile app company, it's clear there must not be a one-size-fits-all way to evaluate development methodologies. Or, if there is, it must be flexible enough to account for changing project inputs. At the conclusion of the research in [2], the most significant realization was that, "there was no one method of evaluation that was always best."

An alternative model for evaluating methodologies is demonstrated by Mitsuyuki, 2017 [4]. They opted to refine the scope of their evaluation to look only at a hybrid between Waterfall and Agile. In doing so, they were able to remove a number of variables, allowing a more focused evaluation of the methodologies. As such, the approaches taken appropriately fall into the categories of *Quantitative experiment* as well as *Quantitative case study*– given the focus on hard metrics such as undetected errors, requirement changes, and number of bugs occurred. They also chose to fix their project inputs to allow for a more controlled experiments and simulations.

Another body of research looked at the project again from a *Quantitative case study, or even possibly an experiment.* In **Figure 2.1**, a strategy for understanding the strength of a development methodology is described by Davis, 1988 [5]. The diagram shows user needs as growing linearly and continuously in order to simplify the model. While this is certainly not the case in reality, visualizing the growth of user needs as a constant helps to better understand the other concepts dependent on that curve. This method introduces a concept of life-cycle properties for a project, such as *lateness, shortfall,* and *longevity. Lateness,* as described by the model, is the measure of time between when a user need has arisen and when the software product is able to satisfy that need. *Shortfall* is the difference in functionality between what is delivered and what is required to meet user needs. The amount of time a system remains a viable option for satisfying user needs is the *longevity.* This approach also introduces two dependent properties, *adaptability* and *inappropriateness.* Based on this strategy, the *adaptability* of a methodology is really how effective, or ineffective, it is at reducing lateness and minimizing shortfall.

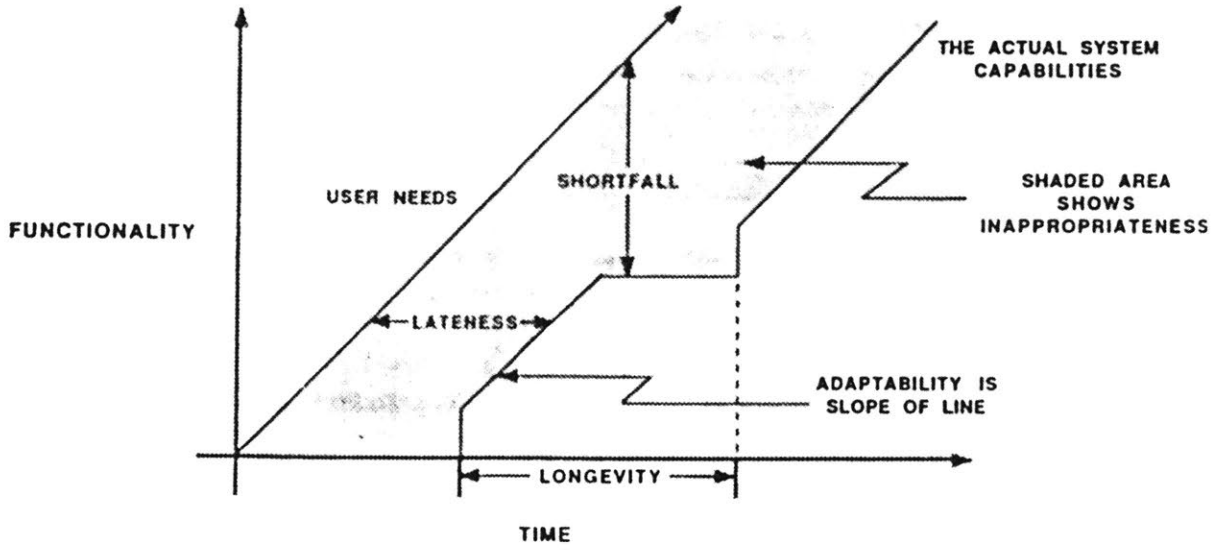

Figure 2.1: *A Basis for Analyzing Software Methodologies - Reproduced from Davis, 1988 [5]*

This could be a useful exercise in understanding how different methodologies stack up against each other. For example, **Figure 2.2** shows a comparison of an *Incremental Development Approach* vs a *Conventional Approach.* Conventional, in this situation, is referring to a waterfall approach.

By lining the methodologies side-by-side in this way, a comparison can be made as to how they perform based on each of the project life-cycle properties defined by the model. One could assume, based on this chart, that an *Incremental Development Approach* is better at addressing user needs than a waterfall approach, because it reduces the level of *inappropriateness* of the system.

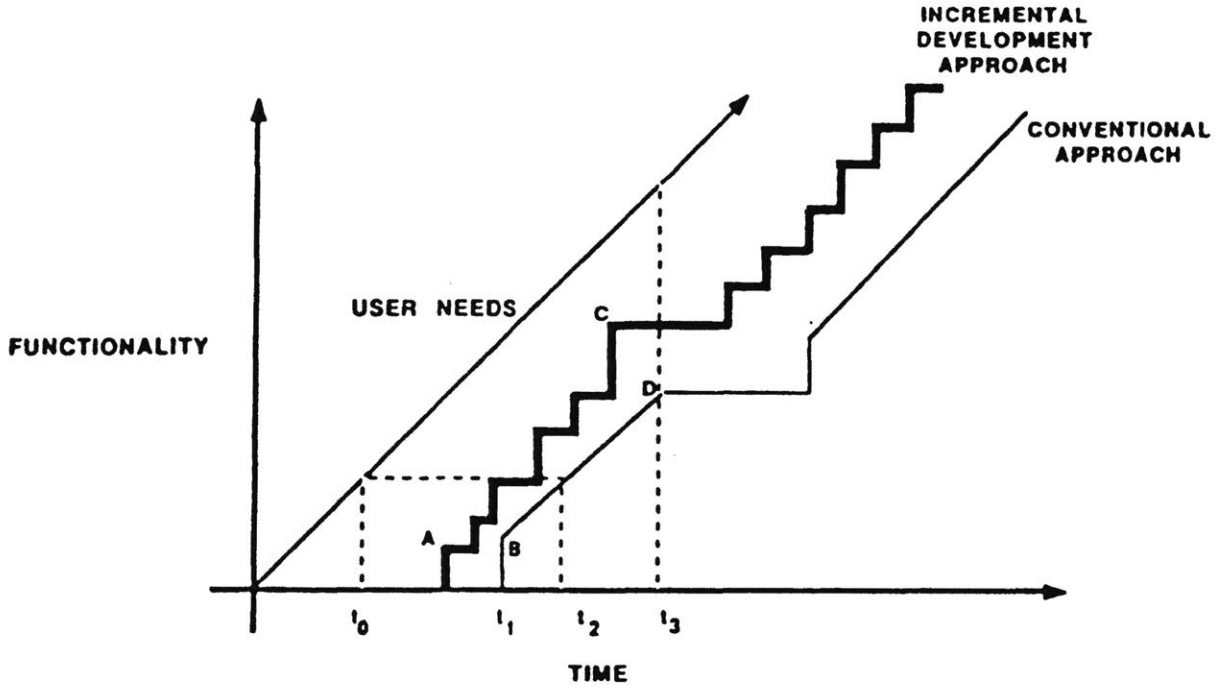

Figure 2.2: *A Basis for Comparing Software Methodologies - Reproduced from Davis, 1988 [5]*

In this approach, both methodologies appear to have almost identical *adaptability* values since the slope of the time versus functionality curve is the same. So, in that regard these two methodologies would be equivalent. The difference between these is that the *Incremental Development Approach* delivers on user needs earlier than the conventional approach. Does this imply that in order for the conventional approach to be as effective at solving user needs, the initial delivery needs to be adjusted? What else sets these two methodologies apart? What other factors should be considered when comparing development methodologies?

	Waterfall	Incremental	Boehm Spiral
STRENGTHS			
Allows for work force specialization	X	X	X
Orderliness appeals to management	X	X	X
Can be reported about	X	X	X
Facilitates allocation of resources	X	X	X
Early functionality		X	X
Does not require a complete set of requirements at the onset		X*	X
Resources can be held constant		X	
Control costs and risk through prototyping			X
WEAKNESSES			
Requires a complete set of requirements at the onset	X		
Enforcement of non-implementation attitude hampers analyst/designer communications	X		
Beginning with less defined general objectives may be uncomfortable for management		X	X
Requires clean interfaces between modules		X	
Incompatibility with a formal review and audit procedure		X	X
Tendency for difficult problems to be pushed to the future so that the initial promise of the first increment is not met by subsequent products		X	X
* The incremental model may be used with a complete set of requirements or with less defined general objectives.			

Figure 2.3: *Comparison of Waterfall, Incremental, Spiral - Reproduced from Sorensen, 1995 [6]*

As an evaluation method, this strategy leaves many questions unanswered. The justification for why a certain methodology is plotted with a specific curve is unclear. What happens if a particular project has different inputs than another? Does an *Incremental Development Approach* perform as effectively when deployed to a team of 100 engineers as it does to a team of 10? How about when the project requirements are well defined but continually changing? This strategy serves as a nice analogy for development teams seeking to select the right development

methodology. It's unclear whether a team is better using methodology A versus methodology B because the impact of many different variables is unclear. Until the various underlying variables are better understood, this approach may not be effective at answering those questions.

In a qualitative experiment, Simão (2011) [8] uses research, documentation, and experts to score Waterfall, Spiral, RAD, Scrum, and XP on Knowledge Areas (KA) within the Software Engineering Body of Knowledge (SWEBOK). The third version of SWEBOK defines 15 KAs [9] (those in bold are KAs included in the study discussed in [8]):

- **Software requirements**
- **Software design**
- **Software construction**
- **Software testing**
- **Software maintenance**
- **Software configuration management**
- **Software engineering management**
- **Software engineering process**
- Software engineering models and methods
- **Software quality**
- Software engineering professional practice
- Software engineering economics
- Computing foundations
- Mathematical foundations
- Engineering foundations

Within each Knowledge Area, the researchers score the methodologies on a series of sub-categories. The resulting output for Spiral is shown in **Figure 2.4**. Similar charts were generated for the other methodologies evaluated:
This approach has a logical decomposition using SWEBOK. An interesting next step would be to take the methodologies and look at them side-by-side. This would be useful for managers to understand which methodology could be most appropriate for their particular situation. This

approach gives good insight into how teams might perform under these methodologies. However, in seeking to understand what makes a methodology better for managing software requirements, for example, this approach does not expose that information as readily. Furthermore, this approach also does not easily give insight into how a methodology might be improved (e.g. adjusted to better allow for management of software requirements).

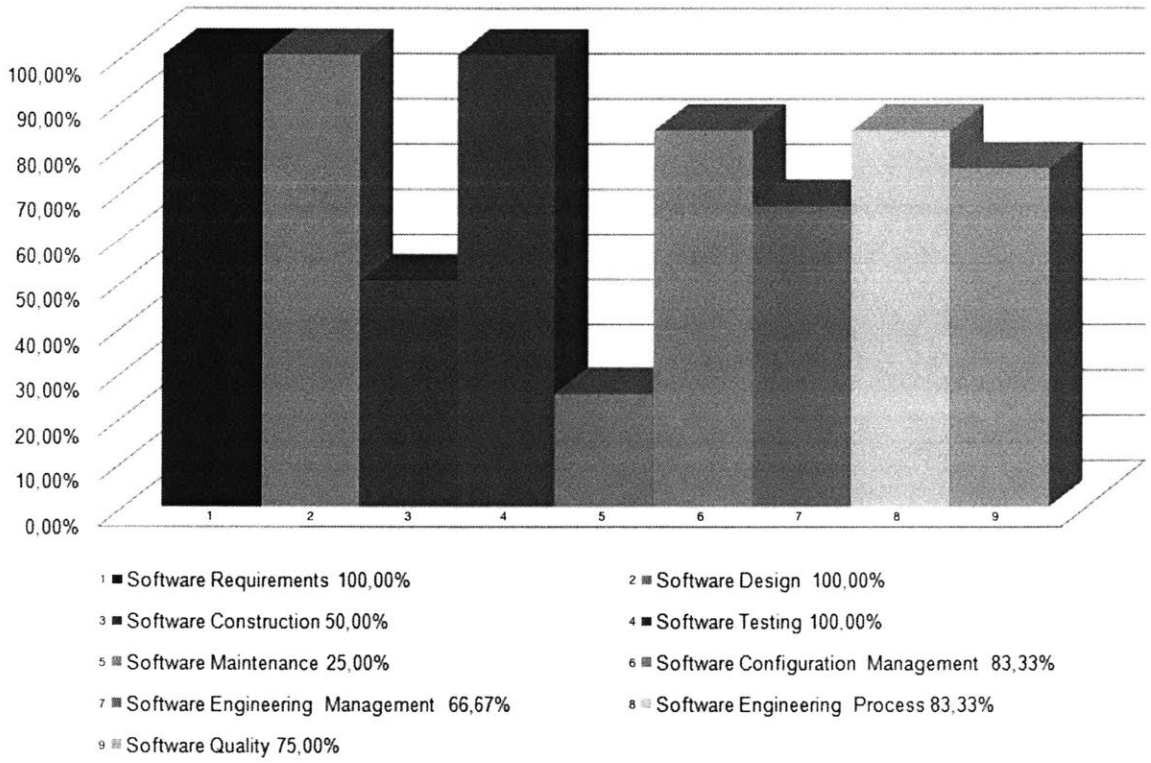

Figure 2.4: *Evaluation of KAs based on SWEBOK for Spiral - Reproduced from Simão, 2011 [8]*

2.2. Comparisons of Development Methodologies

Many bodies of research have been created covering the comparisons of different software development methodologies. Frequently, Waterfall and Agile are two candidates in the race. From that perspective, this thesis is no different. Waterfall and Agile are too important to exclude.

A very common approach to comparing methodologies, is a breakdown of attributes, such as with positives and negatives then population of a matrix as shown in **Figures 2.2 and 2.5**.

		Methodology						
		Waterfall	AUP	Scrum	TDD	RAD	JAD	FDD
Quality Criteria	Requirements flexibility	No	Yes	Yes	Yes	Yes	Yes	No
	Requirements fulfillment guarantee	Yes	Yes	Yes	No [1]	Yes	No	Yes
	Cost estimation	Yes	Yes	Yes	No	Yes	No	Yes
	Cost estimates refinement	No	Yes	Yes	No	Yes	No	Yes
	Validation	Yes	Yes [2]	Yes [3]	Yes	Yes	Yes	Yes
	Quick validation	No	Yes [2]	Yes [3]	Yes	Yes	Yes	Yes
	Focus on customer	No	Yes [4]	Yes	No	Yes	Yes	No
	Understandability guarantee	Yes [5]	No	No	No	No	Yes [6]	No
	Technical debt control	Yes	No	No	Yes	No	No	No
Agility Criteria	Prioritizes added value	No	Yes	Yes	Yes	Yes	Yes	Yes
	Allows partial requirements	No	Yes	Yes	Yes	Yes	Yes	Yes
	Focuses on small teams	No	Yes [7]	Yes	Yes	Yes	Yes	Yes
	Develops minimal viable architecture	No	Yes	Yes	Yes	Yes	Yes	Yes
	Produces minimal documentation	No	Yes	Yes	Yes	Yes	No	Yes
	Relies heavily on customer feedback	No	Yes	Yes	No	Yes	Yes	No
	Susceptible to unforeseen risks	No	Yes	Yes	Yes	No	Yes	Yes

Figure 2.5: *Comparison of Software Development Methodologies - Reproduced from Shaydulin, 2017 [10]*

This approach can be a helpful way to organize observed characteristics of individual software methodologies and see them at a glance [6], [10], [11]. There are three primary ways where this type of comparison fails to deliver adequate evaluation of side-by-side comparisons.

First, in each of the matrix comparisons observed, only binary values were used to determine whether or not a given methodology exhibits a certain desired, or undesired, characteristic. It's much more likely that each of the methodologies exhibit characteristics on some continuous scale. Can one really claim that a waterfall methodology has no *focus on customer* while every agile-like methodology automatically does? Likely, the authors of this research considered this shortcoming. However, to say whether Scrum or Feature Driven Development enables more effective abilities to *focus on customers* would require a deeper understanding of the driving elements that enable this ability. Without that fundamental perspective, this claim cannot be made.

Second, looking at the comparisons of the methodologies in this way gives a snapshot view of what the teams may or may not be able to do well working under a given methodology. But this visualization does little to communicate, or help to understand, what makes a methodology

enable or disable a certain characteristic. This mode of thinking also hinders the ability to see how the methodology might be adjusted to be improved in a certain area. In [6], it is shown that the Boehm's Spiral methodology has the ability to control cost and risk through prototyping, while agile and waterfall methodologies do not have this capability. Is there some fundamental reason why Agile and Waterfall are incapable of employing strategies similar to the Spiral method? This way of viewing does not allow a *next step* thought process. As such, evaluators of methodologies, in this situation, are left to choose between the evaluated methodologies and there is no invitation to adapt and modify to fit the needs of their own project and organizational situations.

Third, the high-level criteria selected for the matrices are complex. There are situational, organizational, product, and other drivers that impact and, in a sense, muddy the waters for making any kind of evaluation of the methodology. Characteristics like *Orderliness appeals to management* or *Technical debt control* are as much characteristics of the teams, managers, customers, and the products themselves. It seems a more accurate assessment would treat the methodologies as a coefficient for each of the reviewed characteristics. But they don't automatically imbue projects with magical properties. A project does not automatically become a technical-debt superpower by adopting a Test Driven Development methodology. What are the fundamental and architectural attributes of a the methodologies that drive the coefficients of success or failure?

These are the questions this comparison strategy raises. How well or how poorly do the underlying properties of a methodology enable or restrict the behaviors that drive effective project execution? Why do the properties enable or restrict the behaviors that drive effective project execution? And what are the distilled properties that drive effective project execution? These are the persisting questions after a review of the existing literature on this subject.

3. Current Methods

3.1. Software Development Methodologies

There are many software development methodologies and strategies which can be categorized in myriad different ways: ideologies, strategies, philosophies, processes, development life-cycles, frameworks, and the list goes on. They are simply referred to in this text primarily as methodologies. Reviewing and evaluating each and every methodology is outside the scope of this research. The specific focus has been limited to reviewing four families of methods: Agile, Waterfall, Spiral, and Set-Based. Agile and Waterfall are commonly considered foundational strategies in software engineering. Spiral exhibits some of the structure of Waterfall but on a more compact scale while also sharing some of the iterative qualities of Agile. Set-Based is new to the software paradigm but is characterized by maintaining and pursuing a set of designs in order to eventually down-select to the ideal choice.

3.2. Waterfall

Dr. Winston Royce first introduced Waterfall in a paper published in the early 1970's [7]. In his paper, Dr. Royce states, "The first rule of managing software development is ruthless enforcement of documentation requirements." This statement sums up the strictness and rigidity of the waterfall method surprisingly well. It is highly dependent on thorough documentation and following a plan. This rigidity is, perhaps, one of the reasons it is still used today in many larger organizations. That same rigidity is also what encourages others to keep their distance–especially those in dynamic and changing environments. Dr. Royce further adds in his paper:

> *At this point it is appropriate to raise the issue of – "how much documentation?" My own view is "quite a lot;" certainly more than most programmers, analysts, or program designers are willing to do if left to their own devices.*

Figure 3.1 shows an adaptation of the waterfall methodology to the development of a mobile application. Each box represents one of the phases of development of the system. The arrows between the phases represent feedback loops that occur at the handoff points from one phase to

the next. Within the waterfall workflow, these feedback loops are the primary point where changes are made. Once the handoff takes place and the project moves along to the next phase, modifications to the work done during prior phases becomes much more costly, in both time and budget. As projects move further along toward completion, the risk of change may be reduced due to fewer unknowns. However, the project becomes can still be somewhat exposed because the risks that remain can have great impact on project objectives if they aren't appropriately mitigated.

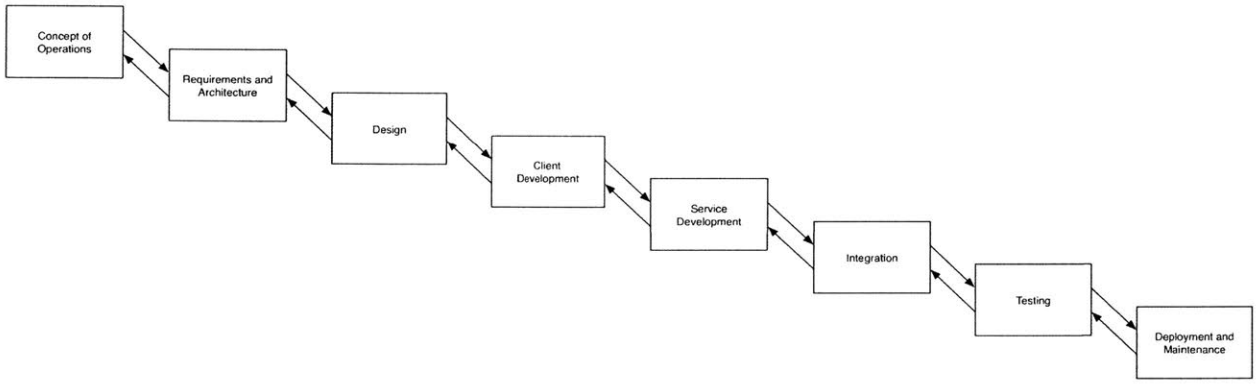

Figure 3.1: *Waterfall Workflow Execution*

Dr Royce showed foresight when he identified the inherent risks with the sequential nature of the waterfall model [7]:

> *The testing phase which occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc., are experienced as distinguished from analyzed. These phenomena are not precisely analyzable. They are not the solutions to the standard partial differential equations of mathematical physics for instance. Yet if these phenomena fail to satisfy the various external constraints, then invariably a major redesign is required. A simple octal patch or redo of some isolated code will not fix these kinds of difficulties. The required design changes are likely to be so disruptive that the software requirements upon which the design is based and which provides the rationale for everything are violated. Either the requirements must be modified, or a substantial change in the design is required. In effect the development process has returned to the origin and one can expect up to a 100-percent overrun in schedule and/or costs.*
>
> *[...]*

However, I believe the illustrated approach to be fundamentally sound.

In his paper, Dr Royce proposes five additional features that could be added to the basic flow previously mentioned to protect against the need for substantial redesign and rework, shown in **Figure 3.2**.

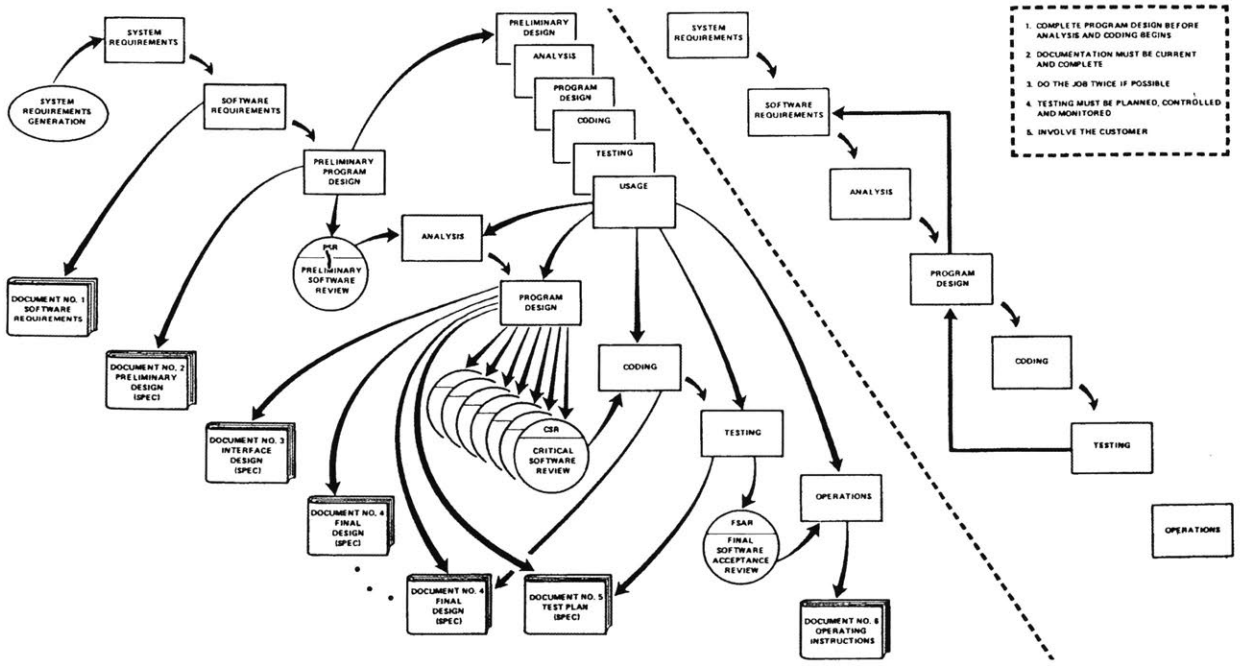

Figure 3.2: Waterfall Concept of Operations - Reproduced from Royce, 1970 [7]

The complex network of processes depicted in **Figure 3.2** could be enough to scare just about anyone away from deploying Waterfall in their project. In many ways, it was frustration with Waterfall's rigidity and complexity that was the catalyst for the creation of other methods, such as Agile. Or rather, the complexity of developing software is not adequately addressed in all situations by deploying the waterfall methodology.

Further clarifications followed on Royce's work over the years. Organizations such as the US Department of Defense created their own Waterfall standards that have been adopted or adapted by software organizations all over the world [12].

Waterfall, in some form or another, is still commonly used today, principally in large engineering organizations. Waterfall is categorized by its sequential order for executing work, a

top-down management style and way of assigning work, and reliance on documentation for maintaining accountability and managing the software development processes.

3.3. Agile

No review of the family of agile methodologies could be complete without a reference to the Agile Manifesto. Created in 2001, the manifesto was a statement made by a small gathering of software engineers frustrated with the state of development at the time [13]. The values addressed in the manifesto are as follows:

> *[...] 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*

In this document, the agile methodology includes other frameworks that are commonly, but not always, considered part of Agile, such as Scrum and Kanban. Agile frameworks are intended to *embrace* changing requirements, even late in development (first principle of the 12 agile principles). In an agile system, team members and key stakeholders (customers) collaborate. The measure for progress is how the system actually functions and there is less reliance on percentage of completion or other similar heuristics. Individuals meet together at regular intervals (often referred to as sprints) to review project progress, reprioritize objectives, and plan for upcoming iterations.

Agile methodologies afford a large amount of flexibility to team members and expect team members to be highly self-motivated and productive. Where Waterfall keeps team members in check through processes, Agile expects team members to self manage. This can result in greater fulfillment for members of a team. But it can also result in greater uncertainty for those responsible for delivering on project goals and objectives. Team members granted authority to make certain decisions they would not be allowed to make in other paradigms. There is also a focus on simplicity and on avoiding unnecessary complexity. It is the art of maximizing the

amount of work *not* done. And team members are given the ability to identify which elements should be kept out of a system, based on the understanding of the first-hand wishes of a customer.

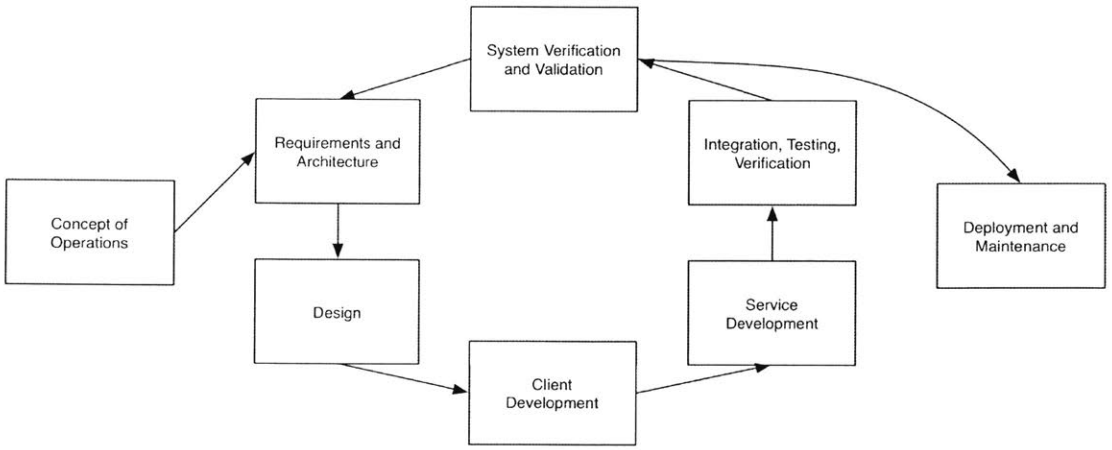

Figure 3.3: *Agile Workflow Execution*

The workflow of an actively executed agile project is visualized in **Figure 3.3**. The project intervals are shown, where elements of work, such as design and development, are carried out in a focused manner during each iteration of the loop. This focuses the team on what will be completed in the near future.

According to the principles of agile software [13], teams are self-organizing and are commonly cross-functional as well. For example, developers, designers, and test engineers could all compose a single team. Among agile teams, members are more likely to volunteer or to be assigned through discussion, rather than having assignments made by a manager. Coordination, clarification, and issues are resolved via face-to-face communication rather than with an appeal to volumes of documentation. Documentation is used when necessary, but is not typically the primary method for maintaining order, communicating ideas, and holding team members accountable for their work.

3.4. Spiral

The spiral development methodology incorporates many characteristics of both waterfall and agile development methodologies. It takes an iterative approach to chipping away at large problems, allowing teams to become increasingly invested as the project moves along. It also has a continual focus on the project goals and technical risks.

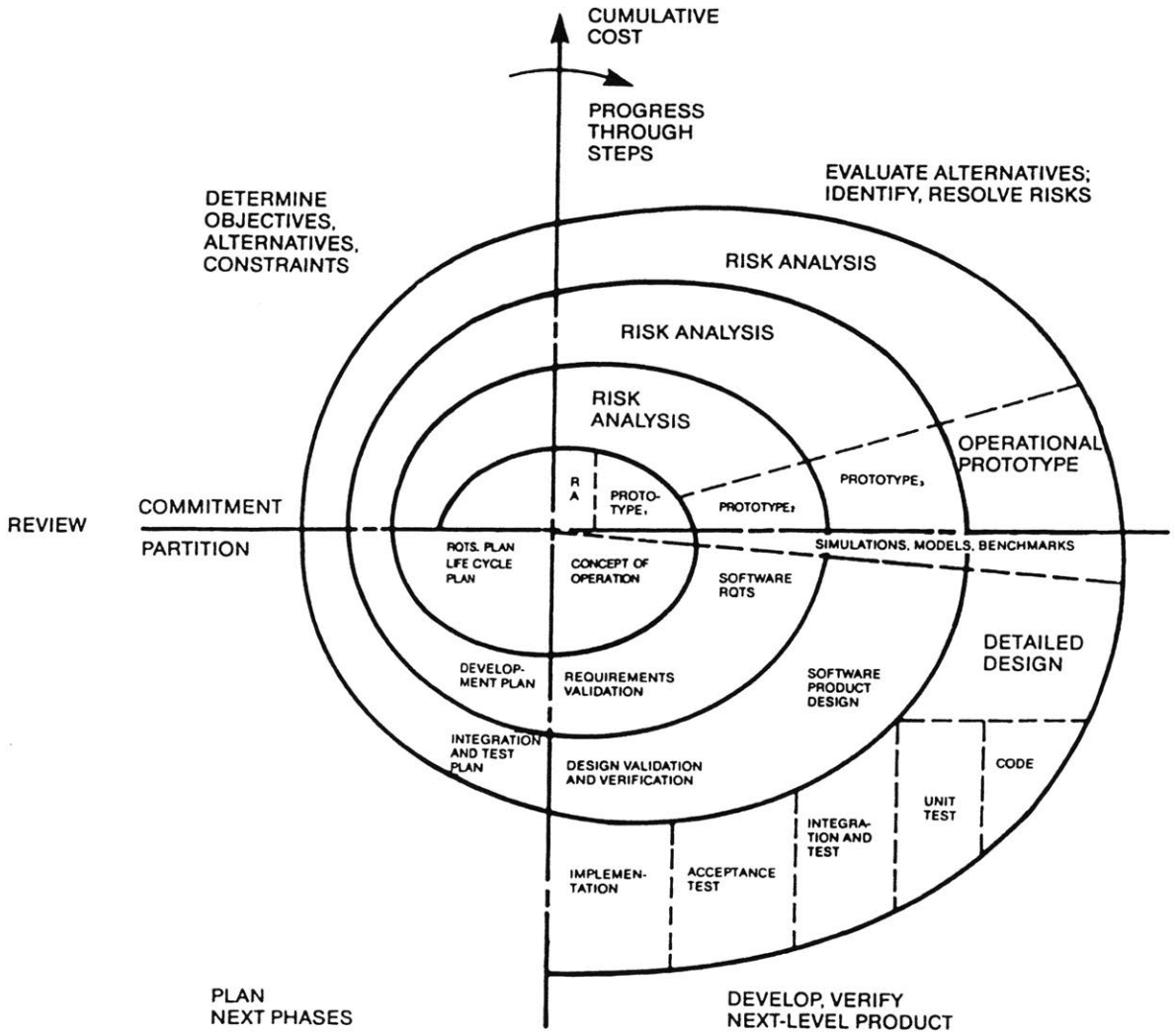

Figure 3.4: *Spiral Methodology - Reproduced from Boehm, 1988 [14]*

In theory, a project starts at the center of the spiral, where the roughest plans and concept of operations are define and prepared. The tenets of the spiral methodology are centered around

managing the risks of the project. At each iteration of the spiral, risks are identified, assessed and mitigation plans defined for the duration of that loop of the spiral. The spiral continues until they're no longer needed.

A typical cycle of a spiral begins by identifying [14]:

- The goals of that segment of the project (functionality, interface design, performance, etc)
- The various options for accomplishing the goals (acquiring, reusing, design option A or B, etc)
- The constraints inherent to each of the options (cost, schedule, complexity, etc)

Each cycle in the spiral model ends with a review by the key stakeholders involved with the development of the product. The scope of the review includes the previous cycle, plans for the next cycle, and any required resources. The purpose is to ensure that all stakeholders are equally committed to completing the next phase of the project [14]. This approach can allow organizations to assume and manage risk in smaller, more digestible chunks. It also allows for software teams to embrace change over time while maintaining a local rigidity within a cycle.

3.5. Set-Based Design

Set-Based Design comes primarily from the hardware design and manufacturing world, particularly from the Naval Ship-Building industry. But it is also used by a number of automotive companies, under a slightly different name, Set-Based Concurrent Engineering (SBCE). Most notably, Toyota, has used this method of engineering quite heavily as reviewed by Sobek, 1999 [15].

Commonly in engineering projects, there is a pressure placed on finding solutions quickly as possible in order to reduce uncertainty. Teams then work on refining those solutions through the duration of the project. Set-Based design works completely in reverse. Set-Based design principles suggest that design decisions should be delayed as long as possible. Rather than seeking to arrive at a solution, the focus is on identifying the weakest options and eliminating them. This allows for concurrent sets of options to be explored asynchronously, since they are independent of each other. This can be done at a micro and/or a macro level of a project.

SBCD is built on three principle phases discussed by Ammar, 2017 [16]:

- Mapping the design space: Create initial requirements that are ranges representing sets of possibilities, rather than single point values.
- Integrating by intersection: Look for intersections of feasible sets while imposing minimum constraints.
- Establishing feasibility before commitment: Narrow sets gradually while increasing detail (staying within sets once committed).

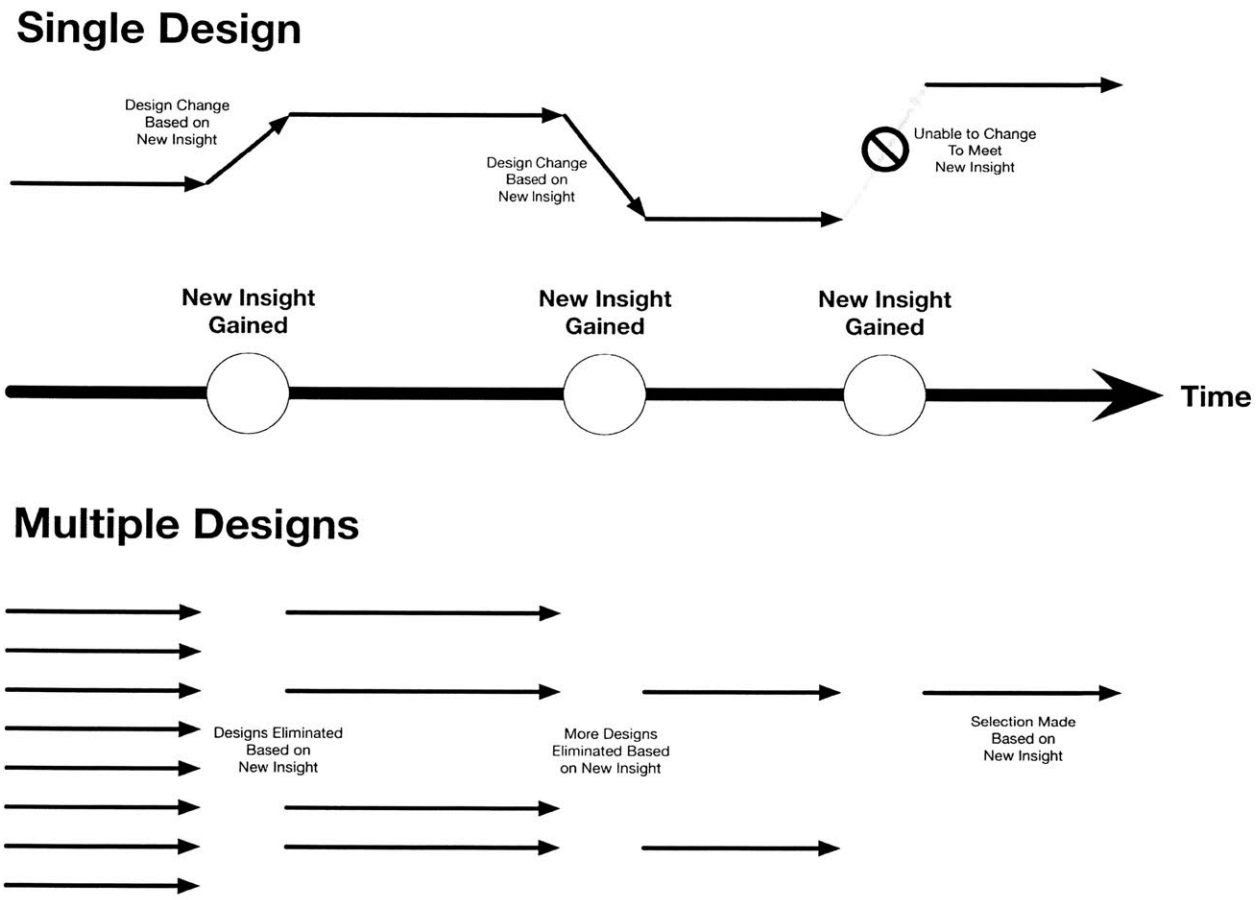

Figure 3.5: *Single Design Approach vs Multiple Design Approach - Adaptations made for this thesis, original published in Scaled Agile Framework, 2018 [17]*

In **Figure 3.5** the concept of Set-Based design is illustrated by contrasting a *Single Design* approach and from a set-based *Multiple Design* approach. The most commonly used approach in software is the serial approach, where a single solution is used and revised as new information is gained. However, as any software engineer or manager knows, inevitably some new

information becomes available too late in development to adjust the plan to take advantage of it. A tradeoff has to be accepted. Either the current solution needs to be reworked to account for new insight or teams must essentially ignore the new information, as the cost to make course corrections at this point is too great. In some cases the latter is not an option and the project must be reworked or abandoned. The multiple design approach shown in **Figure 3.5** works to remedy this concern. By maintaining a set of concurrent designs, each working under the same specifications, a wider range of possibilities can be explored.

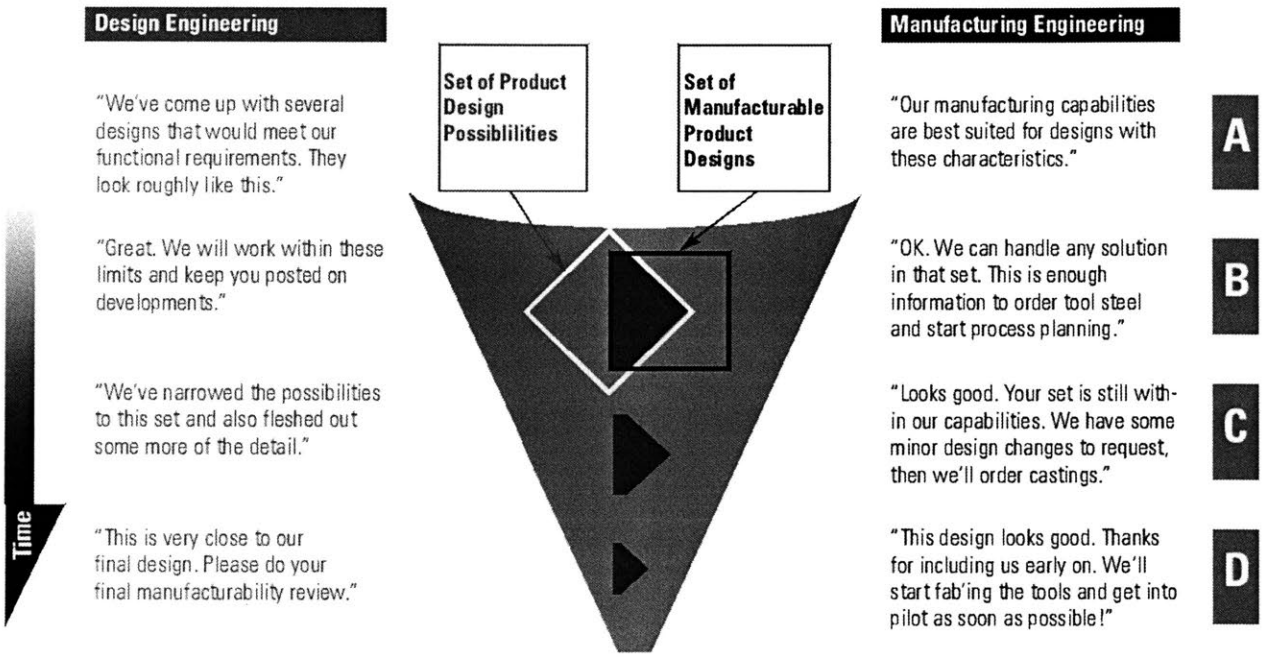

Figure 3.6: *Set-Based Design Principles - Reproduced from Sobek, 1999 [15]*

While well-tested and utilized in the automotive and naval ship-building industries, Set-Based design has yet to be commonly used in the software industry. For reasons currently unknown. It could be because software systems are easier to change than costly hardware systems and therefore incurring the up-front costs of developing multiple solutions in parallel may be impractical. Especially if a possible rework of a single solution could achieve the desired result. Another possible reason could simply be that no clearly defined frameworks exist. Many companies are more focused on building out software systems and are not interested in testing and developing radical and emerging software development methodologies. Regardless of why Set-Based design has yet to see its day for software development methodologies, it represents a distinctly unique alternative architecture to many of the canonical methods currently in

practice. Furthermore, the evaluation of it in comparison to the currently used methods could shed valuable insight into the strengths and weaknesses of all.

3.6 Project Fit

Comparing methodologies is often difficult due to the differences in scope and project objectives. Some methodologies are more prescriptive than others. Other times the methodology may be more applicable at a zoomed-in micro level while others make sense at a zoomed-out, more macro level. In certain cases, the intent of the methodology may be to serve different purposes. As such, identifying how they apply to projects can be problematic. It's no wonder that companies, managers, and engineers have a difficult time identifying which methodology fits their project best. To remedy this, it's important to reduce the methodologies down to their most basic and fundamental building blocks and identify how they match up with projects.

4. Research Approach

4.1. Understanding Project Purpose

Seeking to understand the purpose of a project and how its stakeholders derive value helps to understand the importance of the various aspects of the project. **Figure 4.1** shows the primary function of a software project, which is to meet a stakeholder need through delivery of a

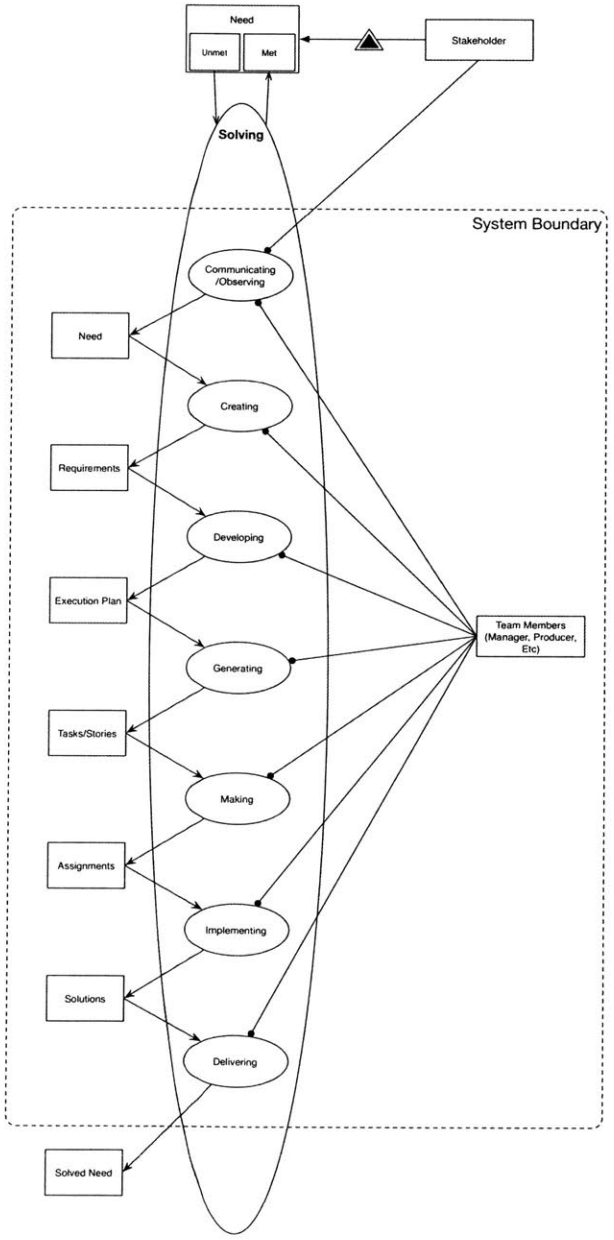

Figure 4.1: *Functional Decomposition of Project*

functioning software product. This is illustrated through a progression of processes executed by team members. This is intended to show the low level actions that work in concert to execute a project. This exercise clarifies the functions and processes at each stage. Beyond this primary delivery of value described in **Figure 5.1**, a number of other supporting sub-processes must be executed. These may include, but are not limited to:

- Identifying and mitigating risks
- Identifying and fixing anomalies in code
- Managing changes
- Managing costs
- Managing schedule
- Testing deliverables
- Sourcing 3rd party software modules
- Sourcing 3rd party services
- Securing application code
- Tuning application performance

This list is not exhaustive, but it helps to better understand what the primary and secondary functions of a software project are and how these functions are translated into value. With this understanding, it is much easier to identify the impact of various project elements. To understand the impact of the deployment of a specific development strategy, it is important to consider not only the primary path for delivery value, but each of the most important sub-processes as well. Another exercise also decouples the value as it's given to a stakeholder from the currently understood mechanisms of the day for delivering that value.

4.2. Building the Model

In order to arrive at a project architecture model that can be used to understand the best fit for a given project's situation, each of the possible architectural decisions must first be identified. These architectural decisions are design decisions of the project with substantial impact on the look of the project. They have a tendency to lock-in other decisions and have particular ripple-down impact on the direction of the project design [18]. These architectural decisions are then mapped to a series of Life-Cycle properties, commonly referred to as Ilities [19]. To achieve the

mapping, the results of these combinations are plotted onto a trade space where a Pareto Frontier can be identified. **Figure 4.1** gives a representation of the flow of this process.

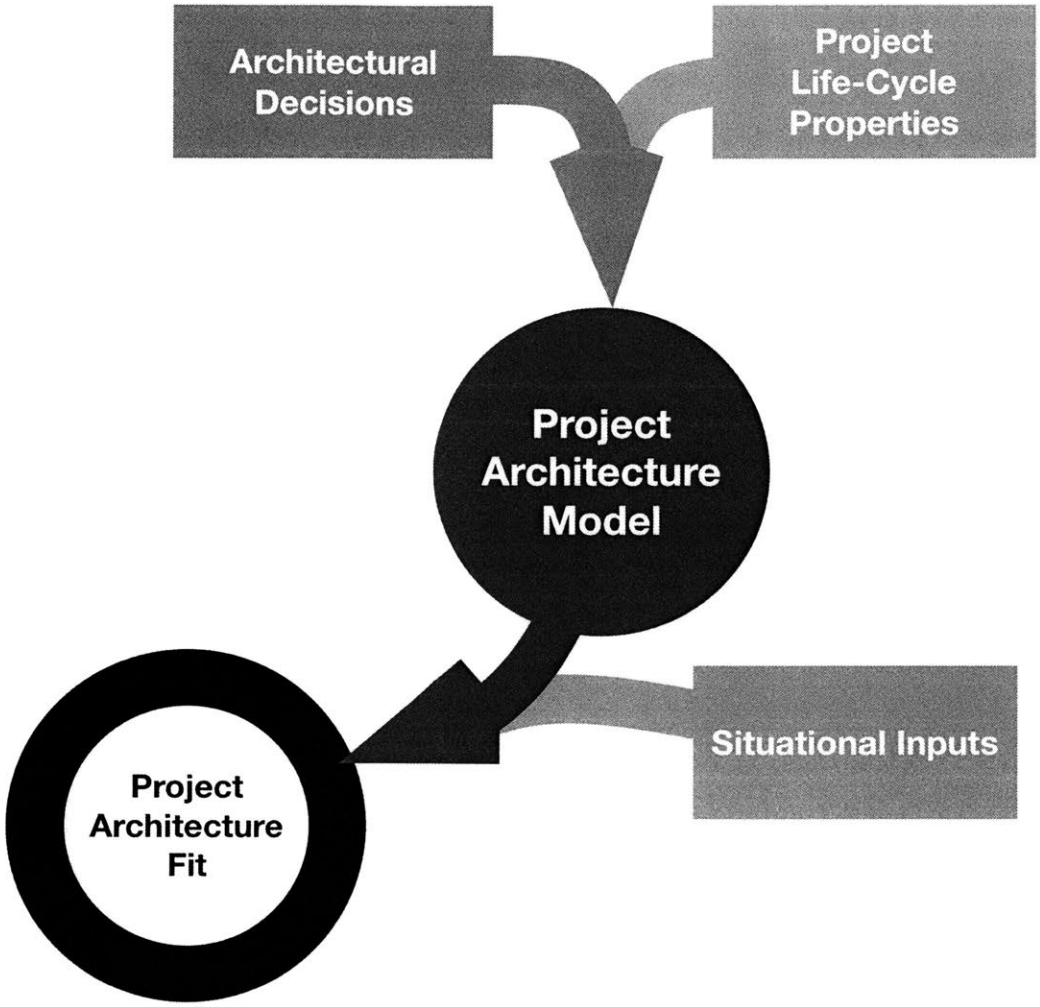

Figure 4.1: *Research Flow*

With a project architecture model, situational inputs can then be gathered from a potential project. These situational inputs are the properties of the project. For the purposes of this research, a series of *toy projects* are used. These toy projects show a range of possible projects, with the purpose of covering a wide range of inputs to tease out the strengths and weaknesses of each of the methodologies. If the model is effective, different project architectures should highlight positive and negative fit combinations of situational project inputs across the various life-cycle properties. For example, it could be expected that a relatively small project with a

limited number of variables but a high degree of uncertainty might perform better against an agile-like project architecture than it would against a waterfall-like architecture when mapped against the ability to adapt to a *changing requirements* life-cycle property. If this is indeed the case, this could indicate a several things:

1. That the claims that agile-like project architectures perform better in small projects with high uncertainty than waterfall-like project architectures
2. That the project architecture model, at least in this instance, did achieve what was expected and provided an accurate classification of the project architecture against the life-cycle properties and project inputs

Inversely, if this is not found to be the case, it could indicate another set of things:

1. That the claims regarding agile-like project architectures could be incorrect
2. That the project architecture model, in this instance, did not achieve what was expected and there may be a crucial flaw in the approach taken

Additional tests will then be performed with other combinations of architectural decisions, life-cycle properties, and situational inputs.

5. Project Architecture

5.1. Project Decomposition: Two-Down-One-Up

A project decomposition is an effective way to visualize and understand a system. A decomposition requires reducing the system down to its basic building blocks. The blocks can then be organized into a hierarchical structure. In the SDM program at MIT, this is commonly referred to as Formal Decomposition. The zeroth level is the containing system, or in this case, the Project. When performing a Two-Down-One-Up Formal Decomposition, the first level of the hierarchy is initially ignored, and the decomposition is performed at the second level. The purpose of this is to allow the first level categorizations or groupings to occur naturally based on the elements that arise from the reduction process. This prevents a premature classification in the first level. In order to maintain a manageable level of complexity at each level, a convention of 7±2 elements at each level is commonly used–though this is not a hard-and-fast rule. The process of decomposing a system is a useful exercise in reverse-engineering an existing system and allows for better understanding of its architectural makeup. It's partially through this decomposition process that insight into properties of architectural elements emerge.

As is done in by Moser, 2015 [20], the project itself can be treated as a socio-technical system, or a system combining both human and technical elements in dynamic interaction. As a complex system, decomposition identifies the project's value. This process provides better insight into the elements that make up a project, which is a prerequisite to understanding their emergent properties. **Figure 5.1** shows the top two levels of a software project formal decomposition. The process of fixing a system boundary around the system helps to clarify which elements will be considered directly as part of the system, and which elements impact the system only externally. The specific justification for which elements are contained within the system boundary and which elements are not, can change from decomposition to decomposition. The rule of thumb used in **Figure 5.1** is to contain those elements that have direct impact on the project and for which the team managing the project has control. There may be a number of projects, individuals, or processes that could affect a project architecture, but are not considered direct contributors and are left outside the boundary.

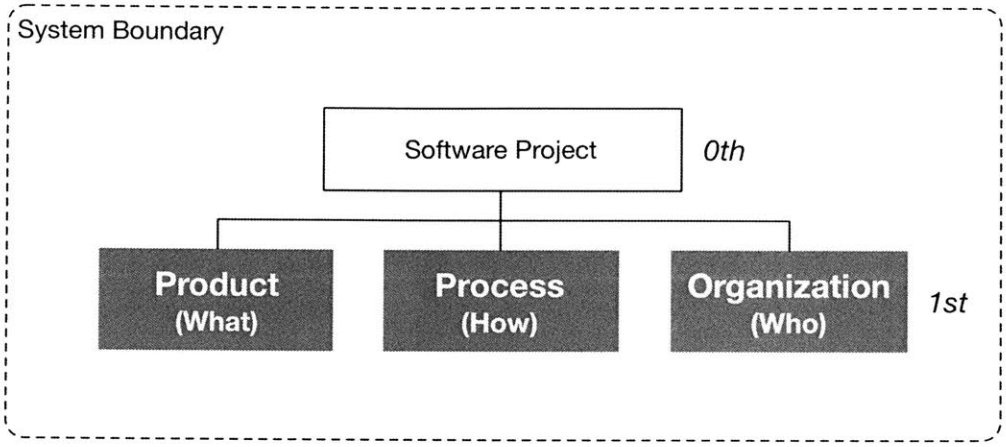

Figure 5.1: *Software Project Decomposition (Level 1)*

5.2. Product Decomposition

Figure 5.2 shows the decomposition of the product branch—or the deliverable. The elements that make up this branch center on those facets of the project that directly impact the product. This includes how requirements are created and how work is segmented, scheduled, and estimated. Each element in the decomposition represents an architectural decision to be made. Some of the decisions are more architectural than others, implying they have a greater impact on the emergent architectural properties of the system.

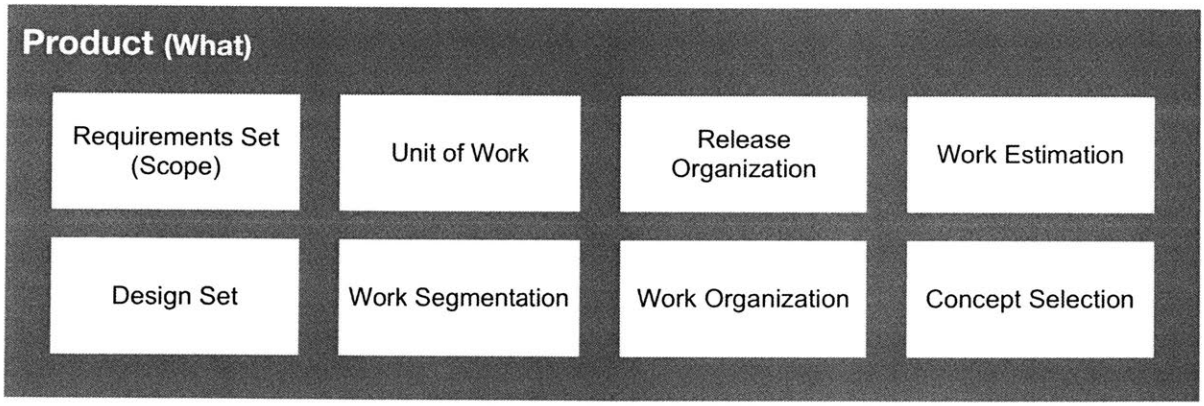

Figure 5.2: *Software Product Decomposition (Level 2)*

A sample product illustrates how architectural decisions might look: A healthcare software system developed for a physician's clinic. Stakeholders may include patients, doctors, nurses, insurance companies, patient families, and so on. Many of these stakeholders may never use the

system, or even know it exists, but they certainly are impacted by decisions made concerning the system development. However, since the team tasked with developing this software does not have direct control over these external stakeholders, they will be kept outside the system boundary. There will also be a number of processes and other software systems inside a healthcare environment needing to coexist with the software system being developed. Each of these external components can be considered in a software architecture and interfaces with them must be considered. However, the management team for the system in question does not have direct impact over them and therefore these systems would commonly be excluded from the system boundary. The decisions at this level determine much around how the Product itself is managed. It is what is decided at this level that drives the architecture of the software project.

5.2.1. Requirements Set

Describes the way the requirements for the project are defined and maintained.

Decision	Option 1	Option 2	Option 3
Requirements Set	Fluid	Fixed	Ranges with increasing specificity

In this decomposition, the *Requirements Set* describes how requirements are managed and maintained. Are they fluid, fixed, or somewhere in between?

In order to build the module of a simple patient record management system, there must be cleanly defined requirements. In the sample project, the system that manages patient records is being replaced. The new system needs to record patient appointments as well as record the results of a visit. The system will primarily be used by the office staff, but individual patients can log on to see their results or export and update their medical and insurance information.

High-level requirements could look as follows (For consistency across methodologies, the shall/ should format is used).

- An office staff member shall be able to manage the appointments for the office.
- A doctor shall be able to enter the summary of an office visit.
- A patient shall be able to access office visit summaries.

As an architectural decision, the *Requirements Set* can take various different forms. For example, requirements can be guiding principles to shape the direction the work takes. They remain fluid to account for incoming feedback from user testing and customers. In other situations, once agreed upon, the requirements remain fixed to allow the plan to be built around them. Requirements may also be viewed as a range with the intention of higher specificity as development progresses.

Ranges with increasing specificity

- A doctor should be able to enter office visit summaries using a method that fits well within their current workflow.
- A patient should be able to access office visit summaries using a method that fits a mobile lifestyle.

These requirements capture the essence of what the user needs but are not entirely explicit. They represent a range of possibilities and give room for exploration. Within these ranges a number of different concepts could be generated. After time is given for exploration, more refined requirements could be established:

Ranges with increased specificity

- A doctor should be able to enter office visit summaries using a stylus on a tablet device.
- A doctor's summary written using a stylus should automatically be transcribed to text.
- A patient should be able to access office visit summaries by having them read aloud via a mobile device.

These requirements fit within the ranges of the original requirements, but they now represent tighter specificity based on some effort of exploration.

5.2.2. Design Set

The number of potential designs considered, initially, or throughout the project.

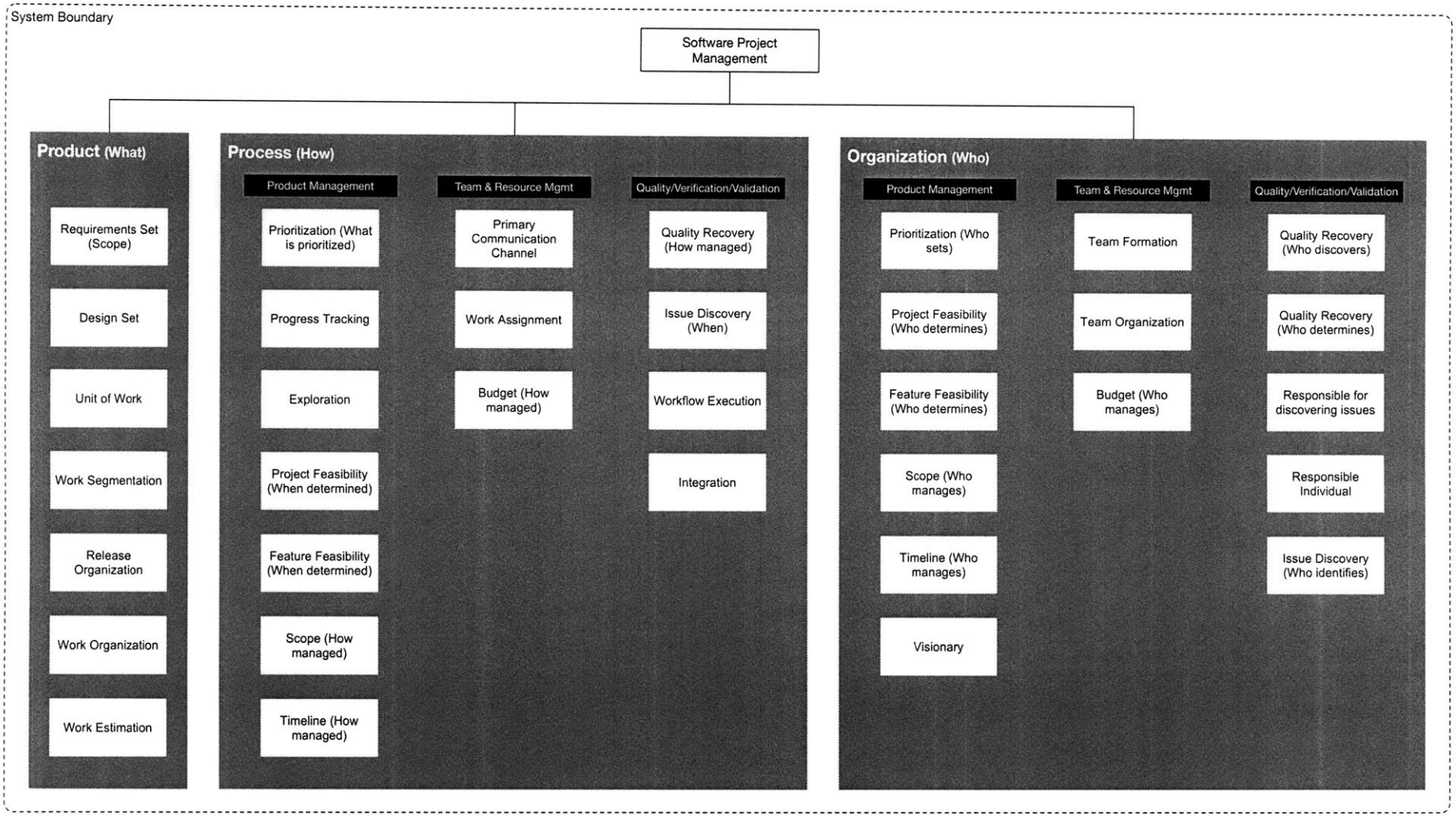

Figure 5.3: Software Project Decomposition

Decision	Option 1	Option 2
Design Set	Single Design	Multiple Designs

A common practice among many methodologies is to reach consensus on a single design solution and work to develop and refine the design throughout the course of the project. However, another possible project architecture could use multiple designs as a way to explore multiple solutions simultaneously. Imagine a situation where a major software company has a need for a particular piece of software to be ready by a certain date. The company has access to substantial resources and missing the particular deadline could cost the company greatly. Multiple development teams could be deployed to work on different designs for the same component in parallel. Developing multiple designs concurrently in this scenario could increase the possibility of finding a solution that will be ready in time.

Another possible use case for a multiple-design architecture could be in a situation where an organization is in search of the optimal solution for a given application. Multiple designs could be explored simultaneously. Evaluations of the resulting designs could then be compared, allowing the organization to identify the best solution to solve the need.

Returning to the sample case of the software application for the physician's clinic, multiple designs could look as follows in practice. The requirement set decision of *Ranges with increasing specificity* could be used here.

Multiple Designs

- Design Option A
    - A doctor should be able to enter office visit summaries using a stylus on a tablet device.
    - A doctor's summary written using a stylus should automatically be transcribed to text.
- Design Option B
    - A doctor should be able to enter office visit summaries using voice to text on a mobile handset.

- Design Option C
    - A doctor should be able to enter office visit summaries using image recognition of a slip of paper with the summary notes handwritten.

Each of these designs satisfy the requirements provided and offer a unique way to solve the need.

5.2.3. Unit of Work

The basic unit of work to be executed.

Decision	Option 1	Option 2
Unit of Work	User-focused story	Task

Units of work are most commonly broken out into user stories or tasks. User stories are focused on the end-to-end resulting functionality for a user. A tasks is commonly written from an engineering perspective. Both come with their strengths and weaknesses. With either case, they should be specific and verifiable. In the case of the patient record management system the same components may be broken out in the following ways:

User-focused story

- As an office staff member, I can add new appointments for patients.
- As an office staff member, I can see upcoming appointments to identify doctor's openings.
- As a doctor, I can record visit results so patients can view them.
- As a patient, I can view my visit results.

Each of the stories is written in an end-to-end verifiable, user-focused way. Instead of focusing on the system, the team members developing the system have to focus on the user and what their needs are. This can protect against losing sight of the *why* when developing features. The same user stories could be written as tasks as well.

Tasks

- Set up a database system.
- Create connectors between application layer and database system.
- Create interface mockups for office staff appointment views.
- Create interface mockups for doctor visit results.
- Create interface mockups for patient visit information view.
- Implement logic to allow staff member to view upcoming appointments.
- Implement logic to allow staff member to create appointments.
- Implement logic to allow doctor to record visit results.
- Implement logic to allow patient to view visit results.

With tasks, there's a greater focus on exactly what is being done in each task. With user stories, there's a greater focus on why the team member is doing what they're doing.

5.2.4. Work Segmentation

How the units of work are organized.

Decision	Option 1	Option 2
Work Segmentation	End-to-end	Phased

Closely related to the individual work units is the work segmentation. Whether features are worked on from an *end-to-end* perspective or whether the work execution is *phased*. If an organization is creating user stories instead of tasks, they're almost certainly going to be also using an *end-to-end* approach for executing the work and this is also commonly paired with a cross-functional team. A task-based method is commonly paired with a phased approach but this isn't always the case. As mentioned, this is closely aligned to the way teams are formed as well, whether as cross-functional teams or teams are grouped by role.

In *end-to-end*, user experience designers, interface designers, software engineers, test engineers, all work on the same features at about the same time or within the same cycle. The designs for the feature are provided to the engineers to implement, and the features are tested in very short iterations. In a phased approach, a more zoomed-out perspective is taken. A whole set of

features (or in some cases, the entire product) start with the user experience and/or user interface teams. Designs are then provided to engineering to implement after which the product is sent to a testing team for testing and validation. When working on a feature end-to-end, if an issue is found by one team member, they can engage the person who just created it. With the phased approach, by the time testing gets their hands on the product, the interface team is already moved on to working on other projects.

This could look like the following in the patient management system:

End-to-End

- A user experience designer creates low-fidelity mockups of the process of a staff member adding an appointment and tests them out with someone else on the team.
- A user interface designer takes the low-fidelity mockups of the feature and creates a high fidelity mockup of the feature.
- An engineer or multiple engineers implement the logic for the feature to function and implement the designs.
- A test engineer tests that as a staff member they can actually add appointments and communicates any inconsistencies back to the team members who created the feature.
- Next, they all move on to the next feature.

Phased

- The interface team creates the mockups for all of the features in the set. Once those have been created and signed off on, they are sent to the engineering team.
- The engineering team then implements the staff user functionality, the doctor functionality, and the patient functionality.
- The test engineers then test and validate that the staff member, doctor, and patient functionality all work as intended. If any of the functions does not pass validation, it's sent back to the team that owns the issue for rework so it can then be resubmitted for testing.

It's worth noting that if multiple design sets are being created, these processes could be happening with multiple teams in parallel for segments of a project. It is also not necessary that multiple designs be carried out from beginning to end of a project. Rather, it explores feasibility of multiple concepts at once to determine the best fit.

5.2.5. Release Organization

How work is assigned to releases.

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Release Organization	Time Bound	Feature Bound	Cost Bound	Time/Feature Bound	Time/Feature/ Cost Bound

There are different ways to segment out work. In some cases, a team may choose to break out work units into individual time-bound segments, commonly referred to as sprints. Sprints aren't the only way teams organize their releases by time. For example, Canonical Ltd, the developer of the popular Linux distribution, Ubuntu, commonly releases twice per year (typically in April and October) [21]. The releases are set for those times and whatever is ready to ship by that time is included in the release. In other situations, a decision could be made to

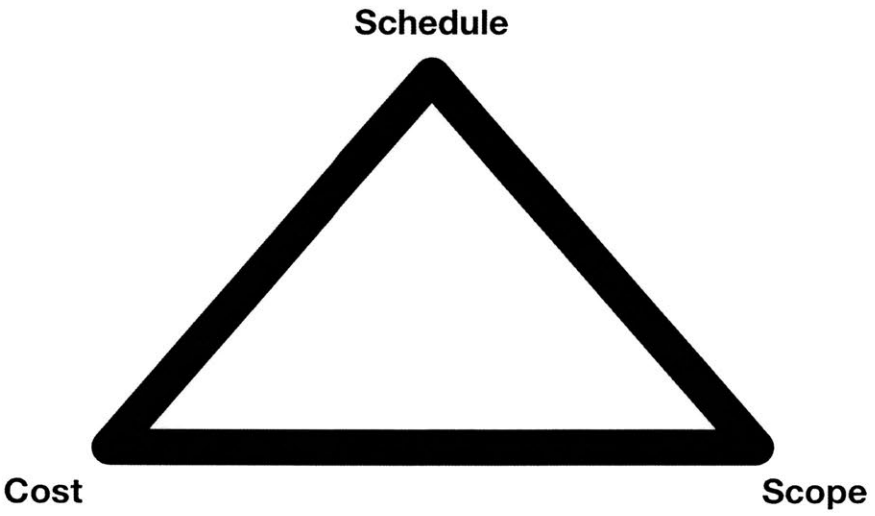

Figure 5.4: *Iron Triangle - Triple Constraint*

focus on milestones, or groups of features and ship when those features are ready. Blizzard, the popular game developer, has historically subscribed to this method. They notably ship their software when it's ready and don't make promises about when that might be. In one of their

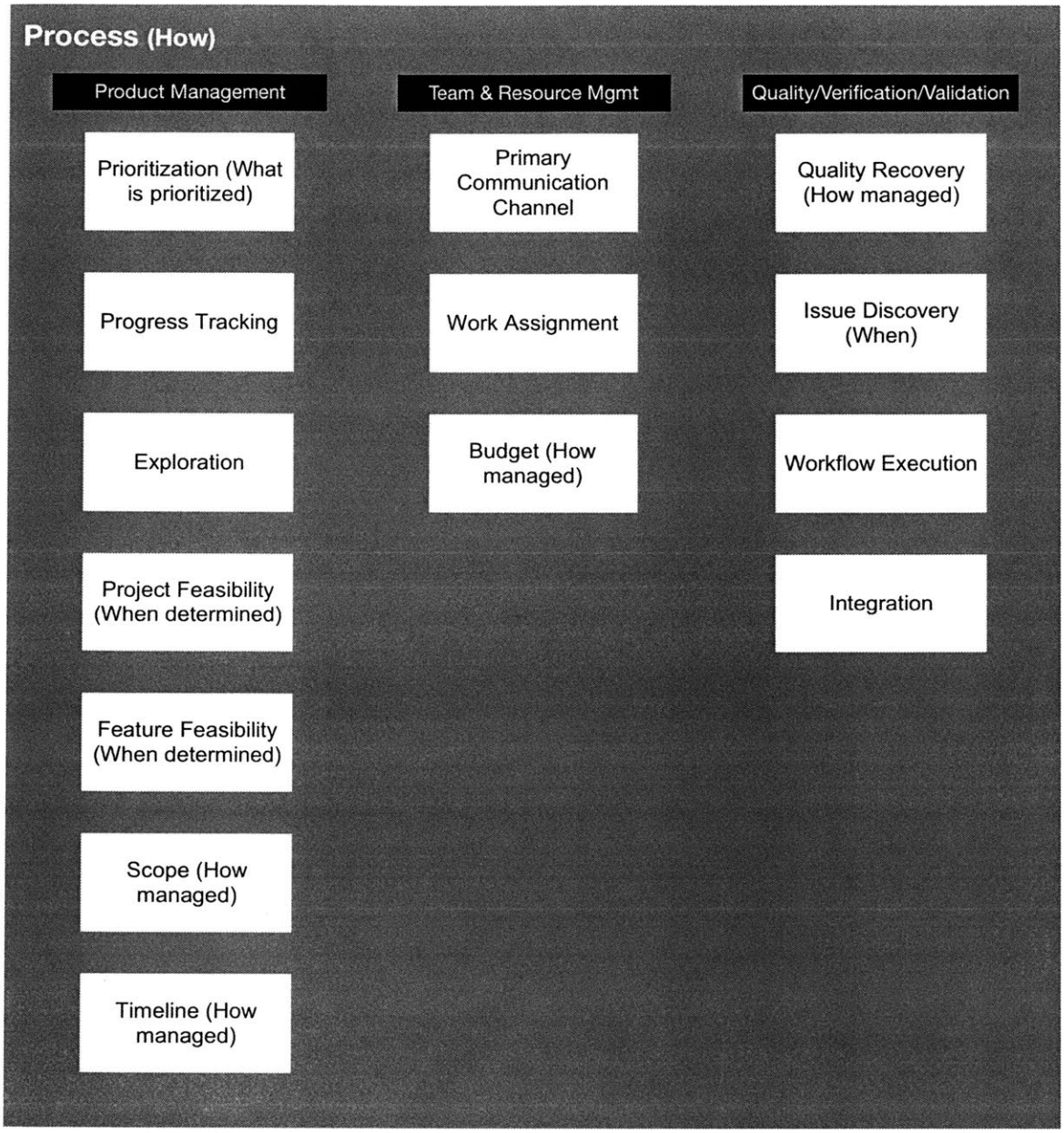

Figure 5.5: *Software Process Decomposition (Level 2)*

core values, they state, "At the end of the day, most players won't remember whether the game was late -- only whether it was great [22]."

Time Bound

- The project will be released once a month. Whatever features are finished and tested by this time will be available to users to user.

Feature Bound

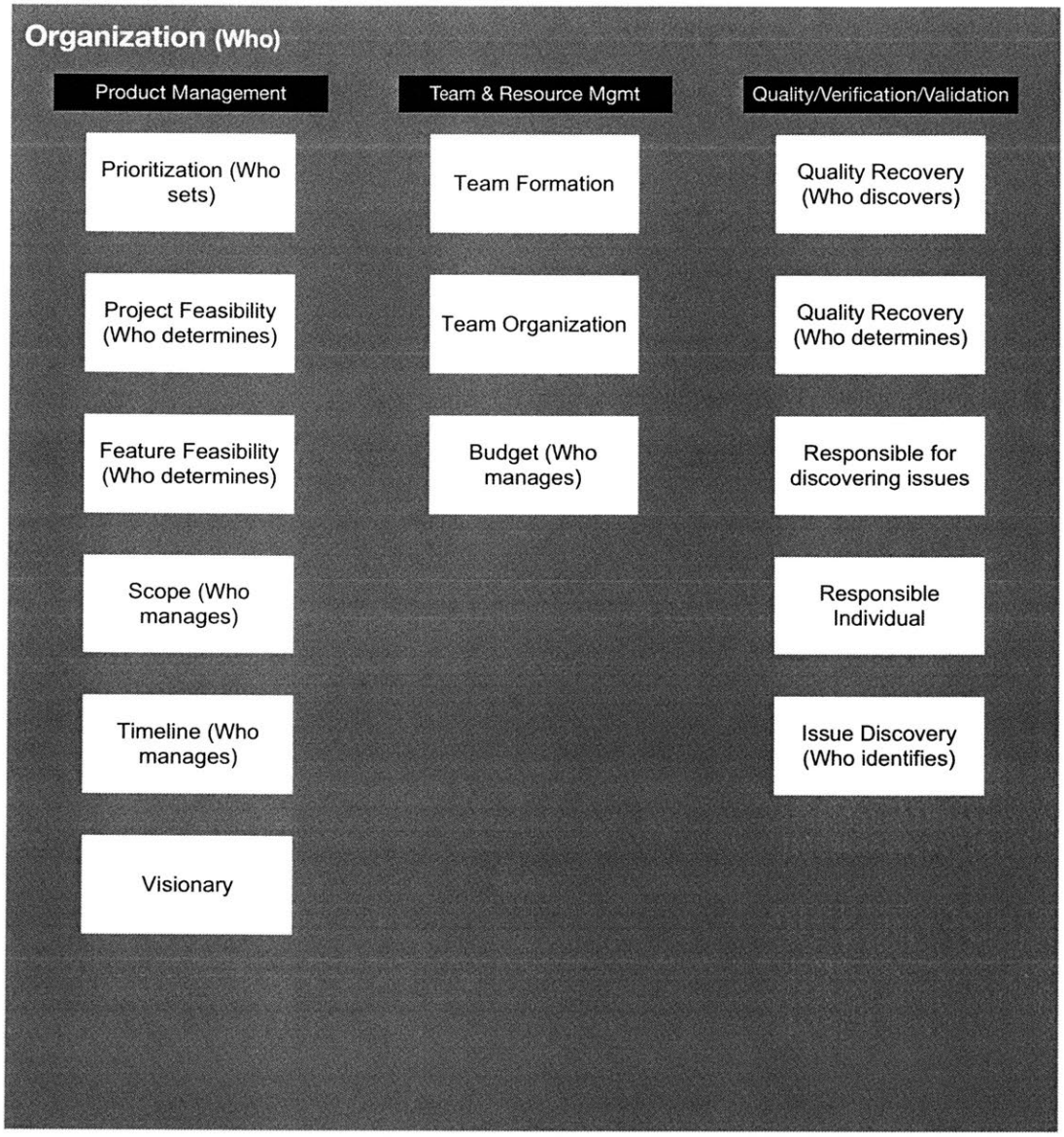

Figure 5.6: *Software Organization Decomposition (Level 2)*

- An initial release must include all functionality for staff members and doctors. All of these features must be included with Milestone 1.
- Patient facing features will be slotted for Milestone 2.

Budget Bound

- A specified budget is set for the project within which the teams need to deliver the patient management system. Whatever the feature set ends up being, it must be delivered within the budgeted amount.

Decision		Description	Option 1	Option 2	Option 3	Option 4	Option 5	Option 6	Option 7
Product (Deliverable)									
Requirements Set (Scope)		Describes the way requirements for the project are defined and maintained.	Fluid	Fixed	Ranges, w/inc specificity				
Design Set		How many potential designs are considered at least initially or throughout the project.	Single Design	Multiple Designs					
Unit of Work		The basic unit of work to be executed.	User-focused story	Task					
Work Segmentation		How the units of work are organized.	End to End	Phased					
Release Organization		How work is assigned to releases.	Time Bound	Feature Bound	Cost Bound	Time/Feature Bound	Time/Feature/Cost Bound		
Work Organization		How product elements to be developed are categorized.	Modular System	Monolithic System	Hybrid System				
Work Estimating		Method for measuring how long a work unit will take to complete.	Difficulty to Velocity	Hours of Effort					
Concept Selection		The method used for down-selecting to the specific product architecture.	Solution Seeking	Non-Solution Seeking					
Process									
Product Mgmt	**Prioritization (What is prioritized)**	What constraint takes top priority.	Deadline	Scope	Stakeholder Requests				
	Progress Tracking	Method for measuring how close to completion the project is.	Units of W completed	Features delivered	Budget consumed				
	Exploration	Built-in mechanisms for handling exploration.	Phased	Continuous	Initial	Not Executed			
	Project Feasibility (When determined)	Which point in the process the project is deemed technically feasible.	Inception	Post-prototype	Not Done				
	Feature Feasibility (When determined)	Which point in the process feature requests are deemed technically feasible.	Reqs Definition Time	Implementation Time	Not Done				
	Scope (How managed)	The method for keeping track of the elements that make up scope.	Requirements List	Project Backlog					
	Timeline (How managed)	How the deadline is controlled and downstream changes are captured.	Date Bound	Scope Bound	Date & Scope Bound	Cost Bound			
Team & Res. Mgmt	**Primary Communication Channel**	Mechanism used for communicating ideas.	Documentation Driven	Meeting Driven	Model Driven				
	Work Assignment	How work gets assigned to team members.	Member volunteers	Member assigned					
	Budget (How managed)	How is the funding disbersed and applied to the project.	Full Project Funding	Incremental Funding	Performance-based Funding				
Quality, Verification, Validation	**Quality Recovery (How managed)**	Recovery process when product does not meet what has been committed.	Continual prioritization	Post production	Phased				
	Issue Discovery (When)	At what point issues are identified (cases where software doesn't work as expected).	Continuous	M.stone/Gate Reviews	Testing Phase				
	Workflow Execution	They general cycle for how work progresses toward completion.	Iterative	Sequential	Top-to-bot, bot-to-top	Concurrent			
	Integration	Which point independent modules/features integrated with the rest of the platform.	Continuous	Periodic	Phased				
Organization Elements									
Product Mgmt	**Prioritization (Who sets)**	The person(s) tasked with setting the priority for work being executed.	Project Manager	Team Members	Customer	Leadership Panel			
	Project Feasibility (Who determines)	The person who determines whether the project is technically feasible.	Subject Matter Expert	Team Members	Project Manager	Leadership Panel			
	Feature Feasibility (Who determines)	The person who determines whether a specific feature request is technically feasible.	Subject Matter Expert	Team Members	Project Manager	Leadership Panel			
	Scope (Who manages)	The person who determines whether a specific request is a part of scope.	Project Manager	Leadership Panel	Customer				
	Timeline (Who manages)	The person who sets the deadlines for project work.	Project Manager	Leadership Panel	Team Members				
	Visionary	Who holds the vision for what the product needs to be.	Product Owner	Project Manager	Customer				
Team & Res. Mgmt	**Team Formation**	How the teams are formed	Self-Organizing	Top-Down					
	Team Organization	The skill-set makeup of teams	Cross-Functional	Functional					
	Budget (Who manages)	The person or entity who controls allocation of funding and resources.	Project Manager	Leadership Panel					
Quality, Verification, Validation	**Quality Recovery (Who discovers)**	Who determines if software product achieves project goals.	Developers	Systems Engineers	QA Team	Users	Project Manager	Leadership Team	Customer
	Quality Recovery (Who determines)	Who determines if work completed deviates from commitments.	Developers	Systems Engineers	QA Team	Project Manager	Leadership Team	Customer	
	Responsible for discovering issues	Who discovers software anomolies (i.e. bugs, not mis-directed project objectives).	QA Team	Project Manager	Users	External Inspector	Team member		
	Responsible Individual	Who is responsible for the overall success of the project.	Product Owner	Project Manager					

Figure 5.7: Architectural Decisions

Time and Feature Bound

- There's an important Healthcare Conference in June. This system needs to be tested and ready for launch by this time. All staff member, doctor, and patient facing features must be ready by the conference. This may take additional resources to ensure this deadline is able to be met.

Time, Feature, and Budget Bound

- The staff member and doctor facing features are required to be delivered by a specified date and is agreed upon for a certain cost.

5.2.6. Concept Selection

The method used for down-selecting to the specific product architecture.

Decision	Option 1	Option 2
Concept Selection	Solution Seeking	Non-Solution Seeking

This decision is closely linked to the *Design Set* decision to select a single design or multiple designs. If option 1 is selected, then clearly only a single design makes sense. However, if the project architecture uses multiple designs, then *Concept Selection* could be solution seeking or non-solution seeking. This could also be thought of as an *Opt-Out* vs *Opt-In* scenario. With solution seeking, the objective is to find the possible solutions to develop. With non-solution seeking as a selection strategy, project decisions are delayed as much as possible in order to ensure the greatest possibility of having all necessary facts before making a locking decision, at which point, dominated solutions are eliminated leaving only the most promising solutions.

5.2.7. Others Product Architectural Decisions

How product elements to be developed are categorized.

Decision	Option 1	Option 2	Option 3
Work Organization	Modular System	Monolithic System	Hybrid System

Method for measuring how long a work unit will take to complete.

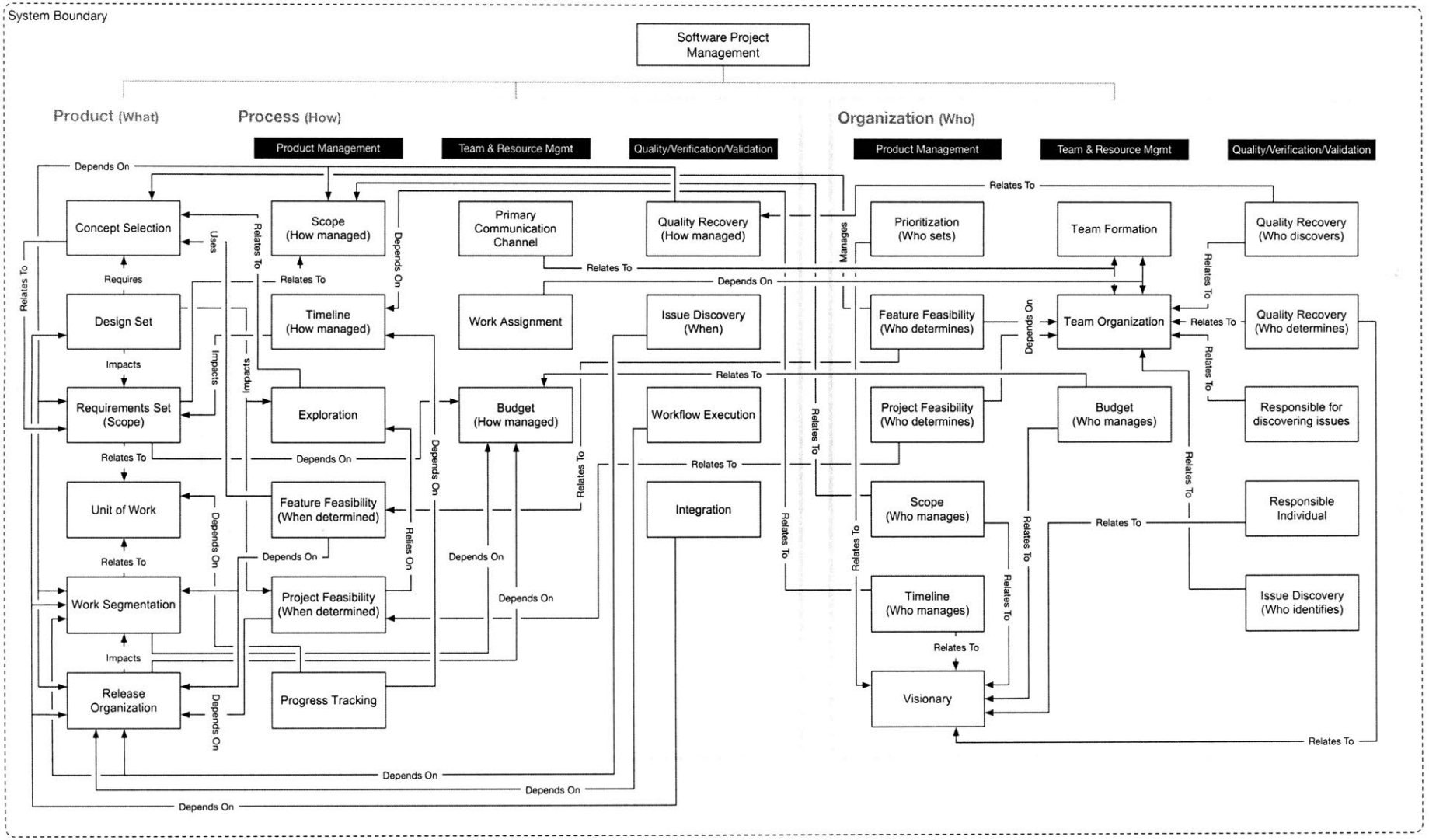

Figure 5.8: Architectural Decision Relationships

Decision	Option 1	Option 2
Work Estimating	Difficulty to Velocity	Hours of Effort

5.3. Process Decomposition

The next layer of decomposition is at the process level. What are the processes that enable the execution of the project? **Figure 5.5** shows these broken out into three individual areas: *Product Management, Team and Resource Management,* and *Quality, Verification, and Validation.* The architectural decisions within the process decomposition are not as readily apparent in the software product, and as such, examples of how they apply to the sample physician's software application are not included. However the individual architectural decisions and their options are shown in **Figure 5.7**.

5.4. Organization Decomposition

The third and final layer of decomposition is at the organization level. This refers to the organizational structure that enables the project architecture. **Figure 5.6** shows the organizational structure broken out. Similarly to the process decomposition, the organization decomposition elements are not readily apparent in the software product. As a result, the specific product results from the sample physician's application are not included here. Individual architectural decisions and their options are shown in **Figure 5.7**.

5.5. Architectural Decisions

Architectural decisions are the subset of design decisions with the farthest reaching emergent consequences for the project [18]. These are the decisions most likely to set one architecture apart from another. As such, these are the decisions most difficult to change down the road. Selecting the right combination of these decisions allows for optimization to the desired scenario. Rather than a binary designation of being an Architectural Decision or not, this designation is more of a spectrum. Some decisions are more architectural than others. The decisions most architectural in nature should be made earlier in the project, reserving decisions that are more design decisions, those with less impact to be made down the road when is most appropriate. Because architectural decisions have such an impact on the success of an

architecture, it's crucial to make these decisions with the right amount of information (i.e. and not make these decisions too soon either). This is much of the motivation of this research–to be able to define the decisions with the most substantial impact on different ilities and situational project inputs, to be able to allow individuals to recognize which decisions they're making and what the downstream effects are of those decisions. To know when a decision they've made will have a locking-in effect on other decisions down the road.

Architectural decisions can be prioritized through a number of means. By looking at how connected their components are to other elements in the system. Decisions involving highly-connected components often have far-reaching emergent consequences, but not always. If a change is made at a highly-connected node, there is a greater potential for ripple down consequences. Much more so than a node with no other connections. **Figure 5.8** shows the Project Architecture decomposition networked to each other. This representation shows how each of the individual decisions interact with one another. In **Table 5.1**, the architectural decisions are then sorted in order by their nodal degree with the most connected decisions at the top.

Other considerations are given for decisions that set one architecture apart from another. If a decision has little to no impact on the overall system, then it is said to be less of an architectural decision and more of a design decision only.

Table 5.1: *Sorted Architectural Decisions by Nodal Degree*

Decision	Degree (In + Out)
Release Organization	8
Team Organization	8
Work Segmentation	7
Requirements Set (Scope)	6
Visionary	6
Quality Recovery (How managed)	5
Concept Selection	5

Decision	Degree (In + Out)
Design Set	4
Project Feasibility (When determined)	4
Budget (How managed)	4
Unit of Work	3
Exploration	3
Feature Feasibility (When determined)	3
Scope (How managed)	3
Timeline (How managed)	3
Integration	3
Progress Tracking	2
Primary Communication Channel	2
Work Assignment	2
Issue Discovery (When)	2
Project Feasibility (Who determines)	2
Feature Feasibility (Who determines)	2
Scope (Who manages)	2
Timeline (Who manages)	2
Team Formation	2
Budget (Who manages)	2
Quality Recovery (Who discovers)	2
Quality Recovery (Who determines)	2
Workflow Execution	1
Prioritization (Who sets)	1
Responsible for discovering issues	1
Responsible Individual	1
Issue Discovery (Who identifies)	1

Nodal degree is not the only factor to consider when determining the architectural impact of a decision.

Certain decisions create hard dependencies with the other decisions. These decisions are tightly coupled with others. With these decisions, it makes less sense to include two decisions that have options that only associate with each other. If A is selected in the first decision, then A must be selected in the second decision. If B is selected in the first, then B must be selected in the second. By combining multiple decisions with this type of behavior, then the results we get from the various combinations will be less insightful. Other decisions have important ripple effects to the rest of the project. In other cases, certain combinations are possible, but should be avoided as they create additional risks. For example, a fixed scope together with a fixed budget and a fixed timeline forms what is commonly referred to as the *Iron Triangle* (see **Figure 5.4**). It is very risky for a development team to take on a project under these terms as projects rarely go according to plan.

An architectural decision should have meaningful emergent properties. The font used in a mobile app, for example, has very little impact on the the development of the app. And, even though it may have a high nodal degree based on the components with which it interfaces, and depending on the architecture, it's relatively simple to change and therefore represents more of a design decision than an architectural decision. Along the same lines, the architectural decision around who is the "Visionary" has a high nodal degree. It's not to say that a design decision doesn't have a substantial impact on the outcome of a product or project. It's just to say that it may not need to be decided as early in the process (i.e. at the architectural level). Selecting the position held by the person who holds the vision for a project, while it touches a large number of other elements within the project, is not a very architectural decision because it's easy to change out and doesn't impact the other decisions in a meaningful way. However, determining whether a budget is to be released based upon hitting a series of milestones versus monthly can have great impact on the other decisions and would therefore be a very architectural decision. This single decision could determine which methodology or type of methodology should be selected.

Given the desire to compare several of the canonical development methodologies, it's also important that given the designation of a specific set of architectural decisions it's possible to describe and differentiate each methodology using this vocabulary.

Figure 5.6 shows a morphological matrix of architectural decisions for project design, with a description of the intent of each decision. The different options for each decision are also shown. The objective of this matrix is to be able to use this as a vocabulary for describing and defining the different canonical software project management methodologies and then to be able to compare them to one another.

5.6. Describing Canonical Methods using Architectural Decisions

One of the main difficulties of comparing one methodology to another, is that each approach uses a different nomenclature or jargon to define their processes and elements. What one methodology calls a user story another methodology calls a task. To complicate the situation, the terms mean something slightly different in each methodology.

This complication can be remedied by defining common denominators or building blocks for strategies that truly represent what is actually taking place in the method. A common understanding can be achieved allowing for cleaner comparisons to be made. The following decisions can be used to describe the different canonical methods. By creating combinations of these options, one architecture can be differentiated from another. These architectural decisions can also be used to articulate other architectures outside of the canonical norms accepted today. This allows for optimization based on the actual needs of the project, rather than a requirement to fit the organization, processes, or product to the specific methodology.

Architectural Decisions

1) Release Organization: How work is assigned to releases.
2) Team Organization: The skill-set makeup of teams.
3) Work Segmentation: How the units of work are organized.
4) Requirements Set (Scope): Describes the way requirements for the project are defined and maintained.

5) Quality Recovery (How managed): Approach for managing how quality is recovered when what is being produced does not meet the standards/specifications for what has been committed.
6) Concept Selection: The method used for down-selecting to the specific product architecture.
7) Design Set: How many potential designs are considered at least initially or throughout the project.
8) Project Feasibility (When determined): Which point in the process the project is deemed technically feasible.
9) Budget (How managed): How is the funding dispersed and applied to the project.

These decisions are translate to the morphological matrix show in **Table 5.2**. This table shows the architectural decisions with their associated options. For each decision, there associating options contribute, in some form or another, to the differentiation of the project architecture and have some sort of emergent or down-stream impact on the design of the project.

Table 5.2: *Condensed Morphological Matrix of Architectural Decisions*

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Release Organization	Time Bound	Feature Bound	Cost Bound	Time Bound / Feature Bound	Time, Feature, and Budget Bound
Team Organization	Cross-Functional	Functional			
Work Segmentation	End to End	Phased			
Requirements Set (Scope)	Fluid	Fixed	Ranges, with increasing specificity		
Quality Recovery (How managed)	Continual prioritization	Post production	Phased		
Design Set	Single Design	Multiple Designs			
Project Feasibility (When determined)	Inception	Post-prototype	Not Done	Continuous	
Budget (How managed)	Full Project Funding	Incremental Funding	Performance-based Funding		
Unit of Work	User-focused story	Task			
Exploration	Phased	Continuous	Initial	Not Executed	
Concept Selection	Solution Seeking	Non-Solution Seeking			

This morphological matrix can be used to define and describe different methodologies. **Table 5.3** is showing an agile-like architecture by highlighting the options within each decision that make the project more agile. These values were derived using documentation on various agile frameworks, the Agile Manifesto [13], and the 12 principles of Agile, and personal experience of how teams are commonly formed within Agile. A few notes on some of the properties selected here:

- Due to the time bound cycles or intervals, commonly referred to as sprints, agile releases are usually time bound. There are situations where a larger release may be reserved until a set of features have been completed. But typically, especially with a continuous integration scenario, the features ready at the end of the cycle are what end up getting deployed and those that are not yet ready get pushed to the next cycle.
- Work segmentation in an agile-like environment has a focus on the end-to-end experience. The work for a feature is executed with members of the team from various disciplines to be able to satisfy accomplish the objective. The team members divvy up the work and segment it in ways that make sense with each other. This contrasts with a scenario where a group of features are all transferred together from phase to phase (design, development, testing, etc) in an assembly line-type fashion.
- A fluid set of requirements hits at the core of much of the motivation for the agile methodology. Other methodologies were born out of the manufacturing use-case, where products move from station to station and at the end are reviewed for quality and then shipped. This is now how most software is built.

Table 5.3: *Morphological Matrix Highlighting an Agile-like Architecture*

Decision	**Option 1**	**Option 2**	**Option 3**	**Option 4**	**Option 5**
Release Organization	Time Bound	Feature Bound	Cost Bound	Time Bound / Feature Bound	Time, Feature, and Budget Bound

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Team Organization	Cross-Functional	Functional			
Work Segmentation	End to End	Phased			
Requirements Set (Scope)	Fluid	Fixed	Ranges, with increasing specificity		
Quality Recovery (How managed)	Continual prioritization	Post production	Phased		
Design Set	Single Design	Multiple Designs			
Project Feasibility (When determined)	Inception	Post-prototype	Not Done	Continuous	
Budget (How managed)	Full Project Funding	Incremental Funding	Performance-based Funding		
Unit of Work	User-focused story	Task			
Exploration	Phased	Continuous	Initial	Not Executed	
Concept Selection	Solution Seeking	Non-Solution Seeking			

The matrix shown in **Table 5.4** describes a waterfall-like architecture. Options have been selected based on information collected from the definitions of the standard by Dr Winston Royce [7] as well as the standard defined by the US Department of Defense 2167A [12]. The justification for the decisions is as follows:

- Waterfall milestones are generally feature bound (or task bound). When the set of tasks, or requirements, have been delivered, the milestone is considered complete. They are not based upon arriving at a certain date or by hitting a point in the budget.
- Because of the way the work is segmented, the teams are organized by function. Projects proceed from analysis to design and from design to coding, from coding to testing, and so on. This form of progression doesn't translate well to a cross-functional team.
- The processing of moving the software product from phase to phase does not lend itself well to fluid or changing requirements either. When a change is encountered, the module, or in some cases the entire code base, has to return to an earlier stage in the process to be reworked.

Table 5.4: *Morphological Matrix Highlighting a Waterfall-like Architecture*

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Release Organization	Time Bound	Feature Bound	Cost Bound	Time Bound / Feature Bound	Time, Feature, and Budget Bound
Team Organization	Cross-Functional	Functional			
Work Segmentation	End to End	Phased			
Requirements Set (Scope)	Fluid	Fixed	Ranges, with increasing specificity		
Quality Recovery (How managed)	Continual prioritization	Post production	Phased		
Design Set	Single Design	Multiple Designs			
Project Feasibility (When determined)	Inception	Post-prototype	Not Done	Continuous	
Budget (How managed)	Full Project Funding	Incremental Funding	Performance-based Funding		
Unit of Work	User-focused story	Task			
Exploration	Phased	Continuous	Initial	Not Executed	
Concept Selection	Solution Seeking	Non-Solution Seeking			

The matrix shown in **Table 5.5** describes the spiral architecture. Options have been selected based on information collected from Boehm's paper introducing the spiral methodology [14]. The justification for the decisions is as follows:

- Spiral releases are generally feature bound. With each iteration of the cycle, there is a very clear set of objectives that should be achieved. These gates are cleared not when a date is hit or the budget reaches a certain amount, but when the items in the scope of the cycle are achieved.
- The spiral methodology does not prescribe whether or not teams should be cross-functional or functional.
- Requirements within Spiral start out as ranges but then increase in specificity with each progressive cycle.
- Quality is recovered with each cycle in the spiral.
- Spiral methodology doesn't prescribe a unit of work or a budget management strategy.

Table 5.5: Morphological Matrix Highlighting a Spiral Architecture

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Release Organization	Time Bound	Feature Bound	Cost Bound	Time Bound / Feature Bound	Time, Feature, and Budget Bound
Team Organization	Cross-Functional	Functional			
Work Segmentation	End to End	Phased			
Requirements Set (Scope)	Fluid	Fixed	Ranges, with increasing specificity		
Quality Recovery (How managed)	Continual prioritization	Post production	Phased		
Design Set	Single Design	Multiple Designs			
Project Feasibility (When determined)	Inception	Post-prototype	Not Done	Continuous	
Budget (How managed)	Full Project Funding	Incremental Funding	Performance-based Funding		
Unit of Work	User-focused story	Task			
Exploration	Phased	Continuous	Initial	Not Executed	
Concept Selection	Solution Seeking	Non-Solution Seeking			

The matrix shown in **Table 5.6** describes the set-based architecture. Options have been selected based on information collected from [23-24], [31-32]. The justification for the decisions is as follows:

- Set-based development methodologies emphasize requirement sets with ranges, multiple designs within a design set, and non-solution seeking concept selection.

Table 5.6: Morphological Matrix Highlighting a Set-Based Architecture

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Release Organization	Time Bound	Feature Bound	Cost Bound	Time Bound / Feature Bound	Time, Feature, and Budget Bound
Team Organization	Cross-Functional	Functional			
Work Segmentation	End to End	Phased			
Requirements Set (Scope)	Fluid	Fixed	Ranges, with increasing specificity		

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Quality Recovery (How managed)	Continual prioritization	Post production	Phased		
Design Set	Single Design	Multiple Designs			
Project Feasibility (When determined)	Inception	Post-prototype	Not Done	Continuous	
Budget (How managed)	Full Project Funding	Incremental Funding	Performance-based Funding		
Unit of Work	User-focused story	Task			
Exploration	Phased	Continuous	Initial	Not Executed	
Concept Selection	Solution Seeking	Non-Solution Seeking			

6. Project Ilities

6.1. Software Product Ilities

Commonly, the concept of *ilities* is used to describe or characterize a system. Many times these are closely aligned to the *non-functional requirements* of the system. However, the scope of these properties goes well beyond the simple binary test of whether the system works or does not work [19]. It's crucial to understand how the system behaves over the course of its life-cycle. There is ordinality to these characteristics and they describe the system performance over time. Included is a list of ilities commonly used in software products.

Intermediate metrics or heuristics can be used to quantify performance over time, such as in the case of reliability, efficiency, and usability. Non-functional requirements are commonly defined based on these properties and system requirements may specify the system functions within a certain failure rate threshold, for example. Core components of the software system are commonly tracked and the metrics can be used as part of the Service Level Agreement (SLA) of the system.

Quality

- How well does a product consistently deliver its primary value?

Maintainability

- What level of effort is required to maintain the software code?
- What level of effort is required to maintain the system?

Reliability

- What is the confidence level that the system is able to perform its intended function(s)?

Usability

- How easy to use is the system? (Often determined through qualitative user studies, but can be measured using certain metrics as well).

Efficiency

- How long does it take to complete certain key tasks using Fitt's, the Steering Law, or many others not addressed here [25]?

Adaptability

- How well does the system adapt to changing needs of the key stakeholders or users?

Availability

- What is the Mean Time Between Failures (MTBF) for key components [26]?
- What is the system uptime?

Security

- How successfully have key components of the system been locked down and protected from attacks?

Portability

- What is the level of effort required to port the system (or its components) to other possible environments or use-cases?

Scalability

- How well does the system scale to account for additional usage?

Safety

- How likely could any loss occur that is deemed unacceptable to key stakeholders [26]?
- What mitigations have been put in place to reduce the possibility of experiencing failures that put mission objectives at risk [26]?

Fault tolerance

- How effectively does the system degrade when it enters a failure mode?

Testability

- How effectively is the system tested, especially when making changes to code or how resources function?

Clearly, not every ility is as crucial to the success of a system as another. And from situation to situation, the relative usefulness or impact of an ility may change, given inputs of project needs. More will be discussed further regarding how project inputs effects the focus on different ilities.

In Engineering Systems, de Weck speaks on the importance of these Life-Cycle Properties and why they've become vital to complex systems [19]. When automobiles were first invented they were little more than horse-drawn carriages with a motor. However, as their usefulness increased and the scope of their application expanded, the systems were expected to perform

additional sub-functions that were not initially part of the requirements. Initially, it was enough that the automobile was able to transport a user from one point to another powered by a motor. It became increasingly important, however, that the automobile could perform this function with reliability and in a predictable amount of time. That an automobile could achieve this without causing bodily harm to parties involved was also key [19].

For software products, the idea that focusing on ilities improves overall system effectiveness is not a new concept. But there are many different elements that contribute to the adequate performance in these categories. It's important to recognize the quality of the engineering team, the budget available to the project, the state of the particular industry for which the system is being built, and many other considerations. The architecture of the project design and project management methodology selected is only one of these contributors, and any correlations to performance against these ilities can, in many cases, be one or more steps removed. As such, it is important to separate out the the impact a particular project architecture could have against software system life-cycle properties. There needs to be a more direct correlation.

6.2. Software Project Ilities

In order to achieve a similar goal as the life-cycle properties commonly used to evaluate performance of software systems, life-cycle properties can also be used to evaluate the performance of the design of a software project architecture. These properties follow the same constraints and principles as their software-product counterparts. They still have ordinality and could have non-functional requirements and specifications that use them as a foundation. However, they are used to describe the performance of the architecture and design of the project, not the system the project yields. Some properties apply in both analogies. However, some don't translate as cleanly.

Responsiveness to change

- How easily can the project accept new requirements and not disrupt execution?
- How manageable is the project plan for the manager when requirements change?
- How well can a manager see the downstream or ripple effects of a change in requirements?
- Does the architecture allow for change?

Team Size Scalability

- How well does the methodology adapt for larger and smaller team sizes?

Scope Size Scalability

- How well does the methodology adapt for larger and smaller scope sizes?

Complexity Scalability

- How well does the methodology perform over a various number of scales?

Scalability

- How easily can the architecture adapt to additional team members?
- How easily can the architecture adapt to additional teams?
- How well does the architecture handle large numbers of requirements?
- How well can the architecture keep track of large number of components?
- How well does the architecture facilitate the visibility of how elements (components, requirements, delays) impact each other?

Primary stakeholder collaboration (customer)

- How well does the architecture allow for input from primary stakeholders?
- Are there opportunities throughout for the primary stakeholders to get a good view of what is happening and provide feedback?
- Is there a mechanism for the feedback/input provided to be worked into the system?
- Are there opportunities for the primary stakeholder to see the impact of decisions?

Ability to identify project risks

- How well the methodology helps to identify upcoming risks.

Ability to mitigate project risks

- How well the methodology helps to be able to create and deploy mitigation plans.

Ability to identify technical risks

- How well the methodology helps to identify technical risks.

Ability to mitigate technical risks

- How discoverable are risks within the architecture?
- How well does the architecture account for risks to project goals?
- How well does the architecture allow for adaptations to account for mitigations to discovered risks?

Feasibility determination

• How well does the architecture allow for the appropriate determination of feasibility?

Ability to trace requirements

• How well the architecture allows for tracking the satisfactory delivery of requirements

Organizational Complexity

• What is the complexity of the organization under the prescribed architecture?

Progress trackability

• How the information flows through the teams/organization based on the project architecture?

• What is the reliability of the flow of information?

Budget visibility

• How much visibility is there into how the budget is being spent throughout the project?

Budget accuracy

• How accurate are budget estimates given the specific project architecture?

Using these ilities, different methodologies and architectures can be compared and contrasted.

7. Tradespaces

7.1. Mapping Architectural Decisions to Ilities

Figure 7.1 maps the previously defined architectural decisions to a list of six different life cycle properties. The groupings of *Product*, *Process*, and *Organization* are retained from the previous diagram. Architectural decisions from each of these areas are included, though the *Product* area is more heavily represented. This is due to the fact that these decisions are the most impactful on the outcome of the software product.

Each option for the architectural decision is scored against each life-cycle property, or project ility. The reduction of the complex project as a system to its basic elements provides a way to apply a score between a life-cycle property and a corresponding architectural property. Representing the elements in their simplest form aims to reduce bias and avoid situational misinterpretations, though neither of these are completely eliminated. The justifications for each of the scores shown is provided in **Appendix A** of this thesis.

		Product														Process														Org
																Q/V/V								T&S			Prod Mgt			T&S
Scale: 1, 2, 3, 5, 8	Architectural Decision	Requirements Set (Scope)			Unit of Work		Work Segmentation		Release Organization				Desig. Set + Conc. Sel.		Exploration				Project Feasibility (When determined)				Budget (How managed)			Quality Recovery (How managed)			Team Org.	
Software Project Ilities	Architectural Decision Choice	Fluid	Fixed	Ranges, w/inc specificity	User story	Task	End to End	Phased	Time Bound	Feature Bound	Budget Bound	Time\Feature Bound	Time/Feat/Budget Bound	SD + Solution Seeking	MD + Non-SS	Phased	Continuous	Initial	Not Executed	Inception	Post-prototype	Not Done	Continual	Full Project Funding	Incremental Funding	Perf-based Funding	Continual prioritization	Post production	Phased	Cross-Functional	Functional
Responsiveness to change		8	1	5	5	2	5	1	8	8	3	2	1	5	3	5	8	3	1	2	3	1	8	1	8	5	8	1	2	3	2
Scope Scalability		1	8	3	3	5	2	5	3	3	2	2	5	5	2	3	2	5	8	5	5	8	1	5	3	2	2	5	3	2	8
Ease of Customer Collaboration		8	2	5	8	2	8	2	3	5	3	1	1	3	8	5	8	3	1	2	3	1	8	1	3	5	5	2	3	5	2
Ability to Identify Project Risk		1	5	8	8	5	5	2	2	3	2	5	8	2	8	3	5	2	1	2	3	1	5	2	3	5	5	1	3	5	2
Ability to Meet Deadline		1	5	8	3	5	1	1	3	2	1	8	2	2	8	3	5	2	1	2	3	1	5	3	2	5	3	3	2	1	1
Ability to Track Scope Progress		1	8	5	5	5	3	3	1	5	1	3	3	5	2	2	1	3	5	3	3	5	1	5	3	2	1	2	3	1	1

Figure 7.1: *Mapping of Architectural Decisions to Life-Cycle Properties*

One important note: This approach has one obvious shortfall of accounting for the relationships between architectural decisions and how the combination of these decisions might positively or negatively impact the project. That said, there is still value in looking at these results because there is much to gain from the combinations of architectural decisions and their impact on

project ilities. The purpose here is not to create a definitive guide on which architecture reigns supreme. Rather, this exercise offers hints and ideas on combinations not yet tried–or how these combinations compare to others that have already been employed many times over. The purpose is, given a project bent on a certain direction, to determine if another combination of choices might yield a better result when a specific outcome is desired.

With the mappings and scoring generated (as shown in **Figure 7.1**) tradespaces are generated to show the intersection between two project life-cycle properties. The methodologies can be plotted to understand how architectural decisions impact the performance of the architectures on their corresponding software projects where they are deployed.

7.2. Tradespace Comparisons: Ability to Identify Risk and Ease of Customer Collaboration

Figure 7.2 shows two important ilities: *Ease of Customer Collaboration* and *Ability to Identify Project Risk*. In this situation, project risk refers to risk that impedes achievement of project goals. Customer collaboration refers to the ability to receive, respond to, and incorporate customer feedback along the way. Each point along the tradespace represents one or more combinations of architectural decisions.

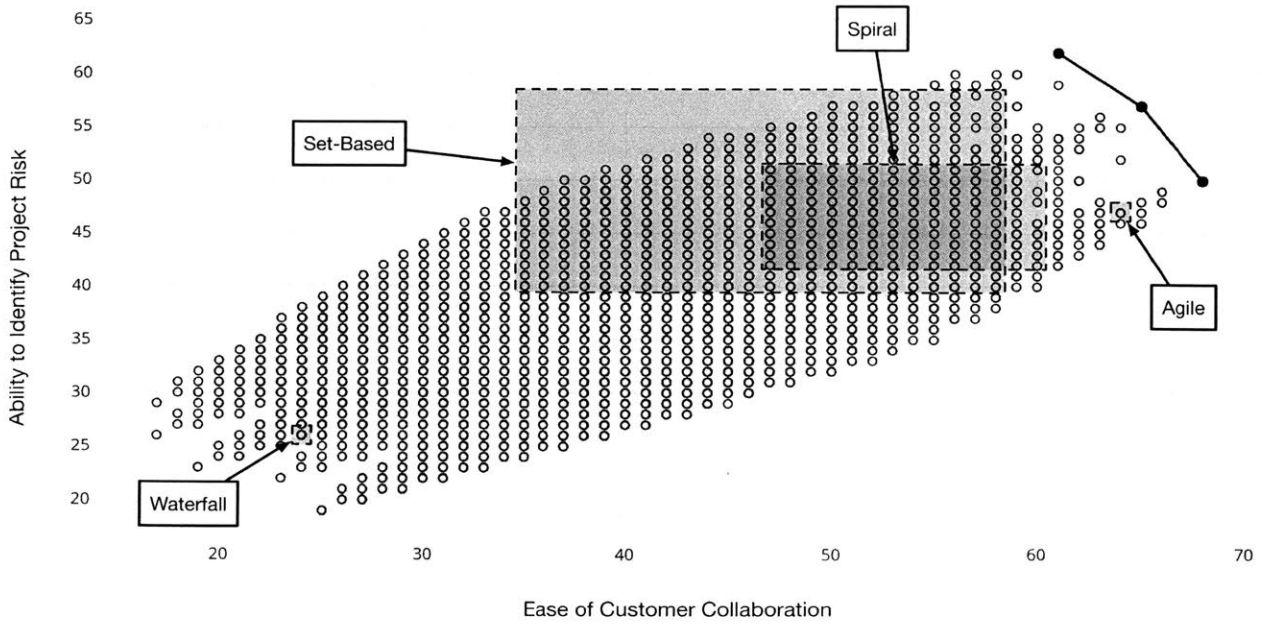

Figure 7.2: Ease of Customer Collaboration and Ability to Identify Project Risk

The combination of these two ilities paints a picture of a methodology that is able to identify risks and adapt to them. A methodology with the ability to identify risks but has no ability to adapt to risk mitigations would not be very effective. Furthermore, this visualization attempts to show how methodologies might perform in a situation where collaboration with a key stakeholder is important. The x and y values are derived by summing the scores of decisions made using the matrix in **Figure 7.1**, which are the scores of each selection mapped back to an ility. In **Figure 7.2**, it is interesting to note where on the plot each of the methodologies lie on the plot. Both Agile and Waterfall are highly prescriptive methodologies. As such, their architectures are well defined and do not represent a wide number of sub-architectures (or variations of architectural decisions) across the tradespace. Alternatively, Set-Based and Spiral are much less-defined (or much less prescriptive) and represent a wider array of architectures, as illustrated by their wider footprint on the chart.

As expected, Waterfall ranks relatively low in the area of *Ease of Customer Collaboration*. It is no secret that many find Waterfall not responsive to change–and this representation supports that notion. Based on the rigidity of this method, this is not a substantial surprise. In contrast, what may be surprising is where Waterfall ranks with regard to *Identifying Project Risk*. Perhaps it is not as well-known that Waterfall, as a methodology, does not perform well at identifying risks to project goals. The fact that architectures within the spiral method rank relatively high on *Ability to Identify Project Risk* is also noteworthy. Risk management is a primary focus within the spiral methodology [14]. Its spot on the tradespace confirms these strengths. Finally, as noted earlier in this thesis, the agile methodology puts a high emphasis on the ability to adapt to customer feedback. This model to validates this claim.

To benefit fully from this research, it's important to follow the path backward through to understand why a methodology scored a certain way. And, as this is an experimental model, it is also important to ask if the results match reality. It may be reasonable to think that Waterfall should score higher in identifying risk, for example. Perhaps the scoring of *Ability to Identify Project Risk* is weighted too heavily toward those selections that allow for changes throughout the life of the project and not enough weight on selections that allow team members to model

and predict pitfalls before they occur. Reviewing justifications in **Appendix A** offers better understanding.

The Pareto Frontier in **Figure 7.2** shows a solid black line connecting solid black dots. These are the optimal architectures for the specific life-cycle properties plotted. This gives insight into the ideal combinations of architectural decisions. **Table 7.1** shows the selections the most optimized architectures have in common (the architectures along the Pareto Frontier). The decisions for *Release Organization* and *Requirements Set (Scope)* are the main differentiators of the architectures along the Frontier and are therefore left open in **Table 7.1**. Following this model, the selections made in **Table 7.1** are the decisions one would make first, combined with either the selections from **Table 7.2** or **Table 7.3**, in order to achieve an architecture along the Pareto Frontier and optimize for these ilities.

Table 7.1: *Optimal Architectures for Ease of Customer Collaboration and Ability to Identify Project Risk*

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Release Organization	Time Bound	Feature Bound	Cost Bound	Time Bound / Feature Bound	Time, Feature, and Budget Bound
Team Organization	Cross-Functional	Functional			
Work Segmentation	End to End	Phased			
Requirements Set (Scope)	Fluid	Fixed	Ranges, with increasing specificity		
Quality Recovery (How managed)	Continual prioritization	Post production	Phased		
Design Set + Concept Select.	Single Design + Solution Seeking	Multiple Designs + Non Solution Seeking			
Project Feasibility (When determined)	Inception	Post-prototype	Not Done	Continuous	
Budget (How managed)	Full Project Funding	Incremental Funding	Performance-based Funding		
Unit of Work	User-focused story	Task			
Exploration	Phased	Continuous	Initial	Not Executed	

A union of the selections from **Table 7.1** and the selections from **Table 7.2** form a single project architecture. Alternatively, given differing situational inputs from a project, selections could be

made from **Table 7.3** instead. For example, an architecture selected from **Table 7.2** would include a *Feature Bound Release Organization* and a *Fluid Requirements Set*. If the selection is coming from **Table 7.3** then the *Requirements Set* would include requirements that are *Ranges, with increasing specificity* as the project progresses and any one of three *Release Organizations: Time Bound, Feature Bound,* or *Cost Bound.*

Table 7.2: *Combinations to Optimize Ease of Stakeholder Collaboration and Ability to Manage Project Risk*

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Release Organization	Time Bound	Feature Bound	Cost Bound	Time Bound / Feature Bound	Time, Feature, and Budget Bound
Requirements Set (Scope)	Fluid	Fixed	Ranges, with increasing specificity		

Table 7.3: *Combinations to Optimize Ease of Stakeholder Collaboration and Ability to Manage Project Risk*

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Release Organization	Time Bound	Feature Bound	Cost Bound	Time Bound / Feature Bound	Time, Feature, and Budget Bound
Requirements Set (Scope)	Fluid	Fixed	Ranges, with increasing specificity		

This model suggests that the decisions from **Table 7.1** will be the most performant with regards to the ilities in question. This methodology would look like a cross between an agile methodology and set-based.

7.3. Tradespace Comparisons: Ability to Track Scope Progress and Ability to Meet Deadline

Another set of ilities to consider are the *Ability to Track Scope Progress* and the *Ability to Meet Deadlines.* Both of these ilities are crucial in the success of a software project. They indicate how well a project architecture lends itself to being able to track its progress, specifically with regard to scope. For example, agile methodologies typically favor being able to respond to change over following a plan [13]. How does this philosophy impact a manager's ability to report on project progress using an agile methodology? How does this philosophy impact a manager's ability to

meet a deadline? **Figure 7.3** shows how agile-like architectures compare to other architectures with specificity to these two properties.

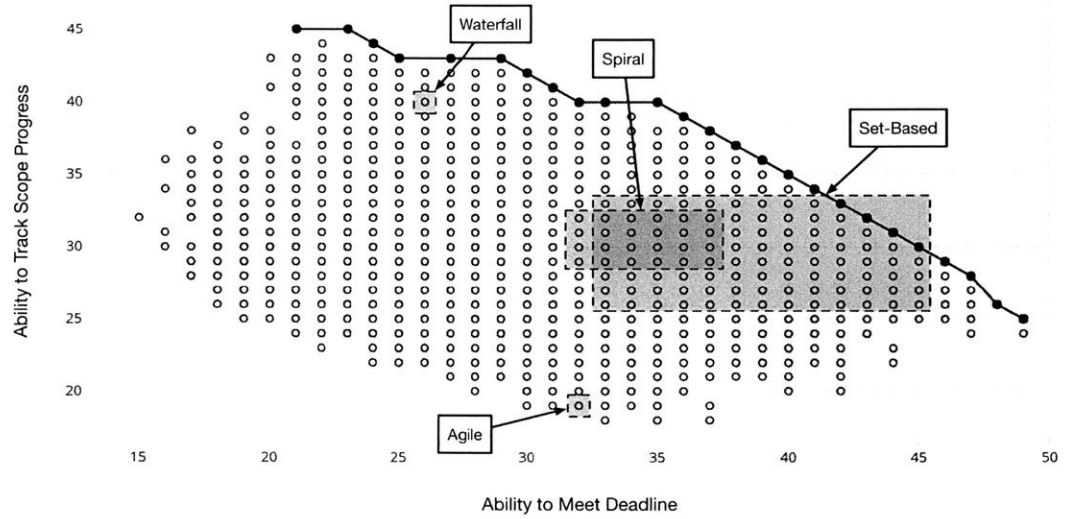

Figure 7.3: *Ability to Meet Deadline and Ability to Track Scope Progress*

Figure 7.3 indicates that Agile does not score particularly well in either category. Based on this model, if hitting a specific deadline is a project priority, a set-based-like architecture may be a better choice.

Table 7.3: *Definitive Decisions to Optimize Ability to Meet Deadlines and Ability to Track Scope Progress*

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Release Organization	Time Bound	Feature Bound	Cost Bound	Time Bound / Feature Bound	Time, Feature, and Budget Bound
Unit of Work	User-focused story	Task			
Budget (How managed)	Full Project Funding	Incremental Funding	Performance-based Funding		

An analysis of the architectures along the Pareto Frontier yields results for the three decisions shown in **Table 7.3**. These decisions are overwhelmingly present in the architectures that are most optimized for meeting deadlines and tracking progress. It is these specific decisions that

this model would recommend locking-in on first. Beyond these, further consideration of other ilities should take place in order to decide which other decisions are important.

Table 7.4: Combinations of Decisions for Optimizing Ability to Meet Deadlines and Track Scope Progress

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Requirements Set (Scope)	Fluid	Fixed	Ranges, with increasing specificity		
Quality Recovery (How managed)	Continual prioritization	Post production	Phased		
Project Feasibility (When determined)	Inception	Post-prototype	Not Done	Continuous	
Team Organization	Cross-Functional	Functional			
Work Segmentation	End to End	Phased			
Design Set	Single Design	Multiple Designs			
Exploration	Phased	Continuous	Initial	Not Executed	

In **Table 7.4**, other options that are optimal in architectures needing to deliver on a deadline and tracking progress are highlighted.

Of the canonical methodologies, Waterfall and Set-Based may be the best choices for optimizing for these two software project life-cycle properties.

7.4. Tradespace Comparisons: Responsiveness to Change and Scope Scalability

Using the same method as before but using a new set of life-cycle properties, the effectiveness of the architectures can be calculated on the tradespace between *Responsiveness to Change* and *Scope Scalability*, as shown in **Figure 7.4**. Unlike the previous tradespace comparison, several of the canonical methodologies find themselves right at, or very near, the Pareto Frontier. This is significant for several possible reasons. Fist, it could be that these two ilities are particularly important properties to teams building software products. As such, designers of methodologies have possibly optimized well for the ability to manage large-scale scopes for projects and the ability to adapt to change. It's quite possible that these methodologies have grown in popularity because they perform well on these very crucial life-cycle properties.

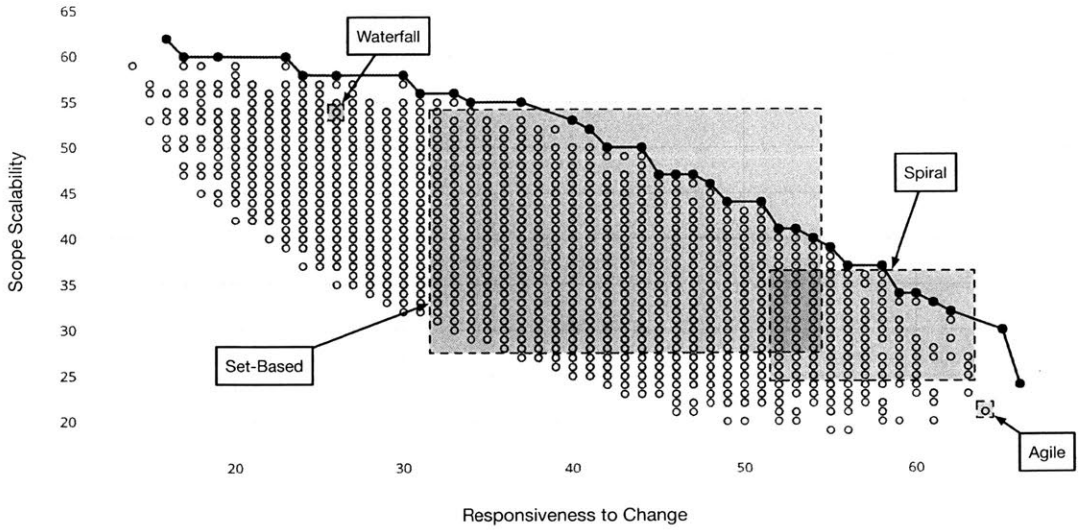

Figure 7.4: *Responsiveness to Change and Scope Scalability Tradespace*

Through and in-depth analysis of the various methodologies and their expected performance in the areas of different life-cycle properties, it is clear that each comes with their strengths and weaknesses. In the outdoor sports industry, downhill skis have properties that make them ideal for certain skiing and snow conditions. Light skis are typically better for touring. A ski with a nice side-cut is better for carving on groomed runs. A nice wide ski is commonly used to float on the top of powder. Each type of ski has their ideal application. Perhaps software project methodologies are not too unlike skis. In consideration of all these different types, there is one category that's an all-mountain ski. These skis are generally not too wide and not too skinny, not too sharp on the cut and not too long. They perform moderately well in all conditions. Based on the tradespace analyses performed here, perhaps the methodology that most closely represents the all-mountain ski would be Set-Based. It tends to perform close to the Pareto Frontier on just about every combination of life-cycle properties. This is most definitely related to its large area on the plot. But further analysis should be done to understand if the architectures up along the Pareto Frontier in one tradespace are the same architectures that appear along the frontier in other tradespaces. This would be an important step to stating definitively that Set-Based design is the all-mountain ski of software development methodologies.

8. Project Situation

8.1. Project Inputs

Showing how tradespace analyses apply to real-world project examples anchors the knowledge gained to something concrete and actionable. To do this, **Table 8.1** defines three individual project examples. Data for projects *A* and *B* comes from Lightning Kite aggregated date, 2015 [27], and data for project *C* comes from Clark, 2015 [29]. This data presents three actual projects (or aggregations of groups of projects) in the private sector and US Department of Defense (DoD). They span a range of budgets, team sizes, project length, and customer requirements.

Table 8.1: *Sample Project Inputs*

Project Input	Project A	Project B	Project C
Number of Team Members	4	25	50
Project Cost	$90K	$720K	$3.8M
Budget Ceiling	Soft	Firm	Firm
Length of Project	4 Months	24 Months	38 Months
Funding Type	Incremental	Incremental	Full Project
Deadline Importance	High	High	High
Possibility of Changing Requirements	High	Moderate	Low
Desire for Customer Input	High	Modeerate	Low
Need for Customer Progress Reporting	Moderate	High	High

Other cases may require additional project inputs. In this particular example, the project inputs were selected because of their obvious links to the project ilities which were highlighted in previous chapters. These inputs represent situational attributes for software projects and identify project needs and priorities. If, in the future, a project requires a different set of ilities mapped to architectural decisions and plotted on tradespaces, a different set of project inputs would likely be desired. As follow-on work, perhaps a more comprehensive list of project inputs could also be selected along with a wider range of ilities.

With the stage of the project architecture models and project inputs now set, rankings are applied to development methodologies based on their fit to individual projects.

8.2. Project Fit to Canonical Architecture: Project A

Project A is a small, mobile application for a new startup venture targeting a single mobile operating system. There is also service work which will allow the mobile application to read and write information via an API. The team is made up of three software engineers and one interface designer. Team members will be performing their own quality testing, validation testing, and interface mockups. Two engineers will be focusing on the mobile application development while the other engineer will be doing the server application code and API development. Because the development project is for a startup company, the requirements are highly volatile and almost certain to change. Also, as part of a startup company, the customer is highly involved and has a strong vision for the problems they want to solve for their eventual customers, and it is not absolutely certain what the final product will look like. The budget has been clearly agreed upon at $90,000 based on the estimates provided by the team and the funds will be dispersed monthly. However, the client is willing to spend more if certain parts of the system become increasingly complex, so the budget is not entirely fixed. The most important aspect of the application development is that it meets the needs of the end customer, and therefore a focus on the feature set is crucial. In addition, time is of the essence. Both parties discuss a 16 week timeline, but no firm due date has been agreed upon.

Which project architecture is best for this particular project situation? Which of the canonical methods is the best fit?

The first place to begin is to identify any of the architectural decisions that have *been decided* based on the project inputs as they are provided. For the purposes of this research, the inputs are considered non-negotiable. However, in a real-world scenario, there are many situations where inputs could be negotiated when justification calls for it. This particular requirement-set will be fluid and, based on the size, the team will be cross-functional. Funds will be available on an incremental basis. The release organization is considered to be time-bound since there is a firm deadline.

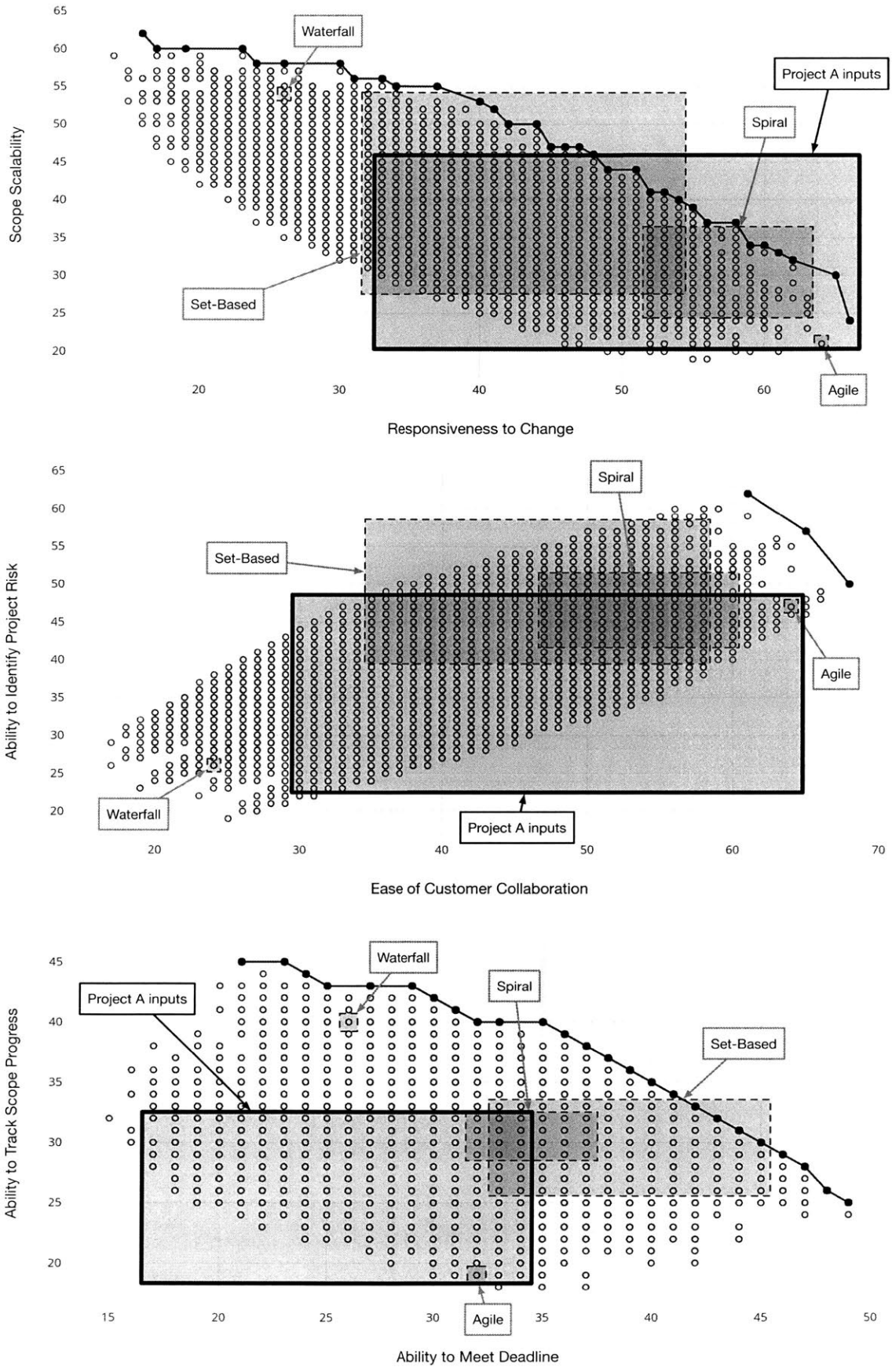

Figure 8.1: Project A Inputs over Ility Tradespaces

Table 8.2: Project A: Architectural Decisions Made Based on Project Inputs

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Release Organization	Time Bound	Feature Bound	Cost Bound	Time Bound / Feature Bound	Time, Feature, and Budget Bound
Team Organization	Cross-Functional	Functional			
Work Segmentation	End to End	Phased			
Requirements Set (Scope)	Fluid	Fixed	Ranges, with increasing specificity		
Quality Recovery (How managed)	Continual prioritization	Post production	Phased		
Design Set + Concept Select.	Single Design + Solution Seeking	Multiple Designs + Non Solution Seeking			
Project Feasibility (When determined)	Inception	Post-prototype	Not Done	Continuous	
Budget (How managed)	Full Project Funding	Incremental Funding	Performance-based Funding		
Unit of Work	User-focused story	Task			
Exploration	Phased	Continuous	Initial	Not Executed	

Figure 8.1 shows the inputs for Project A overlaid on the tradespaces for the six ilities reviewed in this thesis. Project architectures in the bounding box are the architectures that fit the needs outlined by this specific project's inputs. Since, in all three tradespaces, Waterfall falls outside the bounding box, this would not be a great fit for this particular project. Alternatively, Agile is within all three of the bounding boxes, as are Set-Based and Spiral. However, this visualization only shows part of the picture. The visualizations are based on the architectural decisions defined in **Table 8.2**. But not all of this project's inputs map directly to an architectural decision. For example, what do the project inputs say about the importance of each ility? Perhaps an architecture that satisfies the needs of these prioritized ilities would be best.

Given the inputs from *Project A*, ease for collaboration with customer is clearly a high priority. Additionally, the project architecture's capacity to adapt to change and meet a deadline are also important based on the expectation of changing needs and a firm deadline. The ability to track scope progress and identify project risks are of moderate importance. These are not crucial

because the scope is expected to fluctuate. Also, the complexity is low so that risks can be managed by the team without needing an architecture with this particular focus. And finally, since the size of the scope is not particularly large, scope scalability is less important.

On the first of the three charts shown in **Figure 8.1**, project architectures with higher values for *Responsiveness to Change* are better choices for *Project A*. On the second of the three charts, architectures that maximize values for *Ease of Customer Collaboration* are ideal. And on the final of the three charts, architectures that optimize values of *Ability to Meet Deadline* represent architectures with the best fit.

To summarize, there appear to be architectures that would be good choices within Agile, Set-Based, and Spiral for *Project A*. However, Waterfall does not represent a good architecture fit. A deeper analysis of additional project inputs and a narrowing of the specific architectures within Set-Based and Spiral could further narrow the selections and arrive at an optimal architecture for executing *Project A*.

Additionally, there are other architectures with good project fit exist that are not within the canonical methodologies. In fact, it's quite likely that the optimal architecture for *Project A* is not one of the four canonical methodologies reviewed in this research.

8.3. Project Fit to Canonical Architecture: Project B

The needs for *Project B* are different than they were for *Project A*. *Project B* is more complex, has a larger scope, and a greater number of team members. The customer is a global technology corporation. The scope of *Project B* involves building a multi-platform software application with mobile applications and a web client. *Project B* also requires the development of a system to replace an existing, aging system that has become difficult to maintain and is built on outdated technologies. This is the first time mobile applications have been used in this system and context. While the majority of the functions will remain the same, there will be a number of new experimental technologies to be developed. The team allocation in *Project B* is segmented as follows: Android Development Team, iOS Development Team, Web Client Development Team, User Experience and Interface Design Team, Services Development Team, Quality Assurance

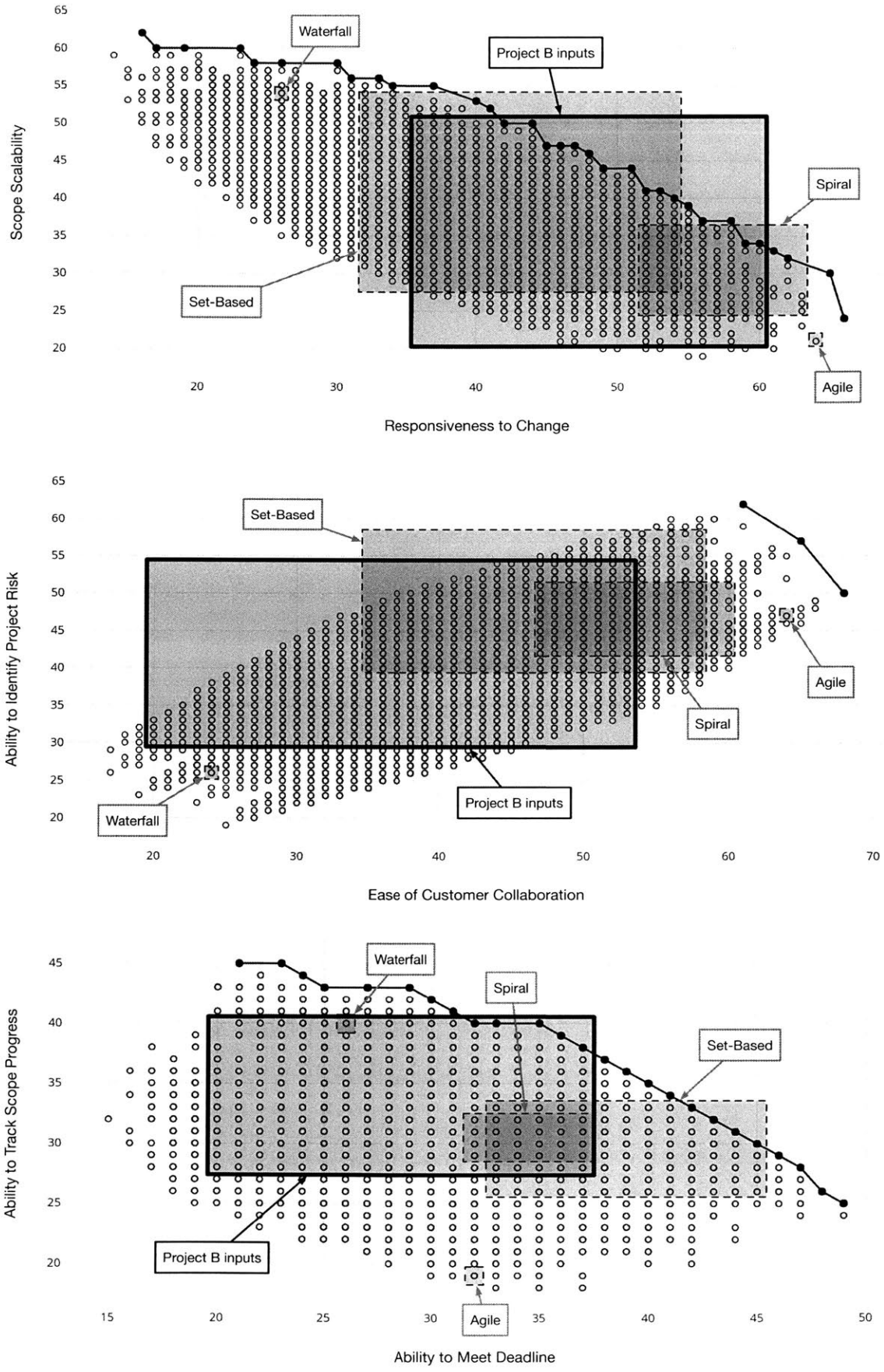

Figure 8.2: Project B Inputs over Ility Tradespaces

Engineers, and Operations Engineers. Many of the team members may not be dedicated to this project full time but will contribute to the project as needed and as availability and priority allow.

Table 8.3: *Project B: Architectural Decisions Made Based on Project Inputs*

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Release Organization	Time Bound	Feature Bound	Cost Bound	Time Bound / Feature Bound	Time, Feature, and Budget Bound
Team Organization	Cross-Functional	Functional			
Work Segmentation	End to End	Phased			
Requirements Set (Scope)	Fluid	Fixed	Ranges, with increasing specificity		
Quality Recovery (How managed)	Continual prioritization	Post production	Phased		
Design Set + Concept Select.	Single Design + Solution Seeking	Multiple Designs + Non Solution Seeking			
Project Feasibility (When determined)	Inception	Post-prototype	Not Done	Continuous	
Budget (How managed)	Full Project Funding	Incremental Funding	Performance-based Funding		
Unit of Work	User-focused story	Task			
Exploration	Phased	Continuous	Initial	Not Executed	

Table 8.3 shows architectural decisions based on situational project inputs from *Project B*. The project goes against the iron triangle shown in **Figure 5.4** of chapter 5. While many in industry recommend against this approach, it is a reality for many projects working on fixed budgets, within a fixed time frame, and with a fixed scope. The results of the architectural decisions from **Table 8.3** are again overlaid on the ility tradespaces (as illustrated in **Figure 8.2)**.

For *Project B*, Agile does not make a good fit. Spiral and Set-Based both appear to have the greatest number of architectures that align with *Project B*'s needs. Waterfall's architecture also does not appear to present a great fit for *Project B*'s situation. To increase confidence, the set of architectures can be reduced for Set-Based and Spiral through continued follow-on work. But at this point, they are the only two canonical methods that appear to be viable options.

8.4. Project Fit to Canonical Architecture: Project C

Project C is an engineering software system for a US Department of Defense development contract. The project is being kicked off after a prototype was created to explore options and determine feasibility. Their requirements are set and unlikely to change substantially. However, they may need to be reworked as new information comes throughout development. There is a firm budget based on an analysis from a research team before the project began, but after the prototype was developed. The organization is extremely risk averse and needs a project architecture that can adequately identify project risks. **Table 8.4** shows the architectural decisions derived from project inputs. The teams have historically been organized functionally, but they're willing to reorganize teams if it will help guarantee the success of the project.

Table 8.4: *Project C: Architectural Decisions Made Based on Project Inputs*

Decision	Option 1	Option 2	Option 3	Option 4	Option 5
Release Organization	Time Bound	Feature Bound	Cost Bound	Time Bound / Feature Bound	Time, Feature, and Budget Bound
Team Organization	Cross-Functional	Functional			
Work Segmentation	End to End	Phased			
Requirements Set (Scope)	Fluid	Fixed	Ranges, with increasing specificity		
Quality Recovery (How managed)	Continual prioritization	Post production	Phased		
Design Set + Concept Select.	Single Design + Solution Seeking	Multiple Designs + Non Solution Seeking			
Project Feasibility (When determined)	Inception	Post-prototype	Not Done	Continuous	
Budget (How managed)	Full Project Funding	Incremental Funding	Performance-based Funding		
Unit of Work	User-focused story	Task			
Exploration	Phased	Continuous	Initial	Not Executed	

Using the architectural decisions from **Table 8.4**, the overlay is created with the project ilities and displayed in **Figure 8.3**. Again, these overlays only represent an initial set of possible architectures based on earlier decisions, due to the project inputs. Further refinement takes place within the boundaries of the decision scope. Inputs from *Project C* should be used to create the

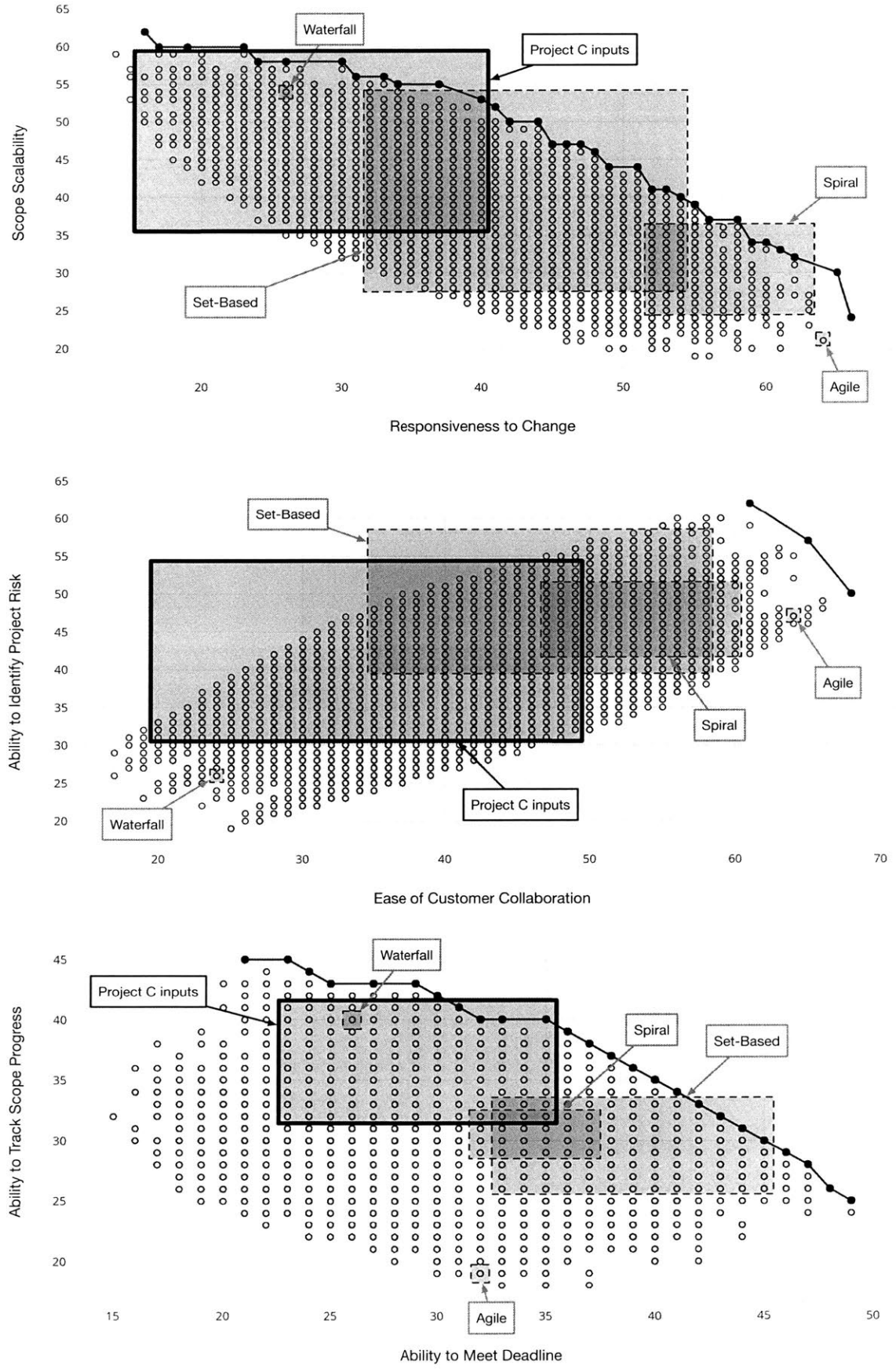

Figure 8.3: Project C Inputs over Ility Tradespaces

further refinement which could allow for selection of a specific methodology, but these are only sample projects. In a real-world example, specific project inputs would further define the needs of the project, ultimately identifying the ideal architecture.

Based on the current set of decisions, it is apparent that Agile is ill-fitted to *Project C*. Neither Spiral nor Waterfall are great choices either. The only methodology suited to serve the needs of *Project C* across all the ilities is Set-Based. While this is promising, additional work is necessary to identify which project architectures within Set-Based should be considered and if they are the same project architectures from tradespace to tradespace. This is a crucial piece of the puzzle for identifying the ideal project architecture.

One substantial take away, aside from all three projects (aside from simply recognizing how the canonical methodologies fit with each project) is that many other possible architectures exist and should investigated. Project managers and other leaders often limit their scope to only the documented methodologies. However, using a systematic approach, other project architectures can be discovered and employed to help increase the probability of successful outcomes for software projects.

9. Conclusion

9.1. Learnings

In summary, there are a number of key takeaways from this research: insights resulting from treating a project as a socio-technical system and reducing it to its basic elements; a new vocabulary to describe and detect methodologies; the concept of life-cycle properties for project architectures; observations from comparing commonly used methodologies and their impact on project ilities; evaluation of architectural decisions and their impact on performance of project architectures in different life-cycle properties; and the possibility of new combinations of architectural decisions forming additional methodologies.

First, when the concept of a software project is thought of as a socio-technical system, it can be decomposed to its basic elements intuitively. Within this research, these basic elements are manifested as the design and architectural decisions that occur when designing a project. This research provides a decomposition of what those decisions are. Too often, managers and other team members make decisions in projects without recognizing they are doing so. As such, a decision made early on may inadvertently result in lock-in on other decisions that follow. This is avoidable, however. Recognizing, for example, that a relationship exists between determining feature feasibility and whether a team is organized as cross-functional or functional gives insight to leaders and aids in understanding the trickle-down effects of their decisions. This empowers project leaders to make decisions deliberately and avoid situations where a manager finds out, too late in the project, that they have limited options because of a seemingly inconsequential decision made at the outset. With the decomposition created here, those involved with software projects can have better understanding as to what decisions exist and what their impact is on project performance.

Using the elements of the decomposed project architecture, a vocabulary is established to describe and differentiate one software methodology from another. Agile methodologies can be described using the combination of architectural decisions that make them agile. Waterfall methodology can be described using the combination of decisions that make it waterfall. And

new methodologies can be described using the combination of decisions that give them a unique architecture. Why is this important? One of the difficulties with comparing methodologies is that they each have a different way of accomplishing similar objectives. Agile uses user stories while Waterfall uses tasks. One is highly prescriptive while another leaves more interpretation to the managers. Comparisons often feel a bit contrived because the methodologies aren't actually comparable. However, by describing them using a common vocabulary of architectural decisions, it's a bit like converting them all to the same currency; it's finding a common denominator. By doing this, the door opens to be able to compare them in the same terms.

Another area that benefits from a new classification system is in life-cycle properties. Life-cycle properties, or ilities as they're commonly referred, have been used for many years to describe software systems. The properties quantify performance and effectiveness of the system. However, there's never been a great way to refer to the performance of project architectures–or software methodologies. This research proposes a list of possible life-cycle properties that can be applied to project architectures, or software development methodologies. Using these terms, a classification and evaluation of project architectures takes place. Managers and team members would be better able to select the methodology that best fits their specific situation and priorities.

With decomposition to basic architectural decisions, life-cycle properties are mapped and scored while minimizing the influence of biases. In addition, the number of questions insulates the experiment from the influence any single decision has.

Calling back to the original hypothesis of this thesis, there are many different project architectures to still be employed and tested. Instead of just a single possible methodology that showed promise or value in being further understood, many new project architectures emerged. In many cases, they exhibit only incremental differences from other methodologies, however, others could combine architectural decisions in ways not yet tried.

For software engineers and project managers, the conversation regarding which specific methodology to use on given project usually stays within agile and waterfall families. However, Spiral and Set-Based both show great promise. Across ilities, architectures that fit the spiral or set-based classification consistently showed on or near the Pareto Frontier. In addition, both methodologies typically showed up in the middle of the Frontier. This indicates that there was a good mix between the two ilities. This could change as the methodologies are refined since currently both of these architectures are represented as ranges, rather than being highly specific like Agile or Waterfall. Additional research in this area could answer these questions.

There is another unexpected concept discovered in the research. Traditionally, it is assumed that there is a one-to-one relationship between a project and a project architecture. However, different architectures could be applied to different segments, phases, or sections of a project. Or different methodologies could be applied at different *zoom-levels* of a project. For example, Set-Based design seems like a good candidate for being applied at a macro level. It provides ways to protect against risks and it helps explore different ways to deliver needs to customers and meet deadlines. However, it does not prescribe work execution in many of the product-related architectural decisions. Conversely, agile-related methodologies could be applied very well at the micro level. The two methodologies could complement each other if used on the same project.

9.2 Shortcomings of the Model

While there is value in the approach taken here project architecture and ility mapping model, it is also important to understand the shortcomings of the model. The life-cycle to ility mappings are qualitative and could contain embedded biases and/or misinterpretations, the impact of multiple decisions in concert is not expressly modeled and therefore not well understood, and there is not currently a great way to track more than two ilities on the same graph. The results of this research are also extremely difficult to validate.

With regards to possibility for embedded biases: This research ventures to be transparent with respect to where decisions were made that may reflect the author's past experience. The section of the research with the greatest risk of being influenced by external factors is in the

architectural decision to ility mapping. Experts in the field have been engaged for review, but this only reduces, not eliminates, the possibility for bias. A list of justifications for each architectural decision to ility score has been included in Appendix A of this thesis for transparency into the reasoning behind each score.

Another limitation of this research approach as executed in the model is that it does not capture how architectural decisions might work together (or against each other) when combined. The approach does not venture to model these certain physical laws or otherwise that result from the combination of multiple decisions. For example, it's not impossible to imagine that a combination of decisions exists, perhaps using *Work Segmentation, Team Organization,* and *Quality Recovery,* where together they have greater impact on the project than just by summing their ility scores. It is also plausible that another combination of options of the same decisions work to counteract each other in a way that's not represented by summing their scores. This seems to be a real world example of the quote by Aristotle: "The whole is greater than the sum of its parts [30]." This phenomenon is not captured by the model in this research.

The model uses tradespace analysis to show the expected top performing architectures across a set of two life-cycle properties. This makes for quick interpretation, but in most situations, project designers consider more than two life-cycle properties in their projects. Therefore, a method that only considers an intersection of two properties does not paint a complete picture and the onus is on the individual consuming the data to put together the pieces of the puzzle. This could lead to missing important pieces of information or not finding optimal scenarios.

There needs to be some way to validate the findings of this research. One problem is that simply looking at the successful results of projects where certain architectural decisions were made could be misleading. In a certain set of cases, very effective leaders and team members could still have favorable outcomes, even if the methodology selected for their software project is not the best possible fit, given the impact of other variables. In other situations, less effective leaders and team members could have unfavorable outcomes even if the methodology selected for their project is the best fit, also due to variables outside the scope of the project architecture. In an ideal scenario, an experiment could create a set of conditions where project inputs remain

constant across multiple projects, including personnel and project complexity and size, allowing the testing of different project architectures. This would give more insight. However, most projects, especially complex and those with large scale, take time to execute. Teams are also unlikely to *experiment* on projects with a large budget given the possible risks.

9.3 Future Work

Through this research, other thoughts, ideas, concepts, and questions emerged. Some of these would require future research and are contained here.

A number of the sources reviewed and considered in this research looked at other industries to understand the approaches they might try in the software industry. Researchers looked at medicine, education, and others [2]. This gave useful insight into the types of complexities other industries face and how they overcome them (or *not* overcome them, in many cases). It is possible that some of the approaches taken in this research could be applied to other industries.

As previously mentioned, one of the difficulties of the approach used in this thesis is the possibility of bias in the architectural decision to life-cycle property scoring. How could the research be insulated from this bias? One obvious way is to engage a wider range of experts to provide additional insight into the scoring of these properties. By leveraging the wisdom of these experts on a series of very specific options, the impact of bias could be minimized. Another way to increase the integrity of the results would be to replace some of the ilities tracked by qualitative metrics to use quantitative metrics, such as the Oli de Weck's work to compute system complexity described in [19].

Other work could include:

- Tagging architectures in tradespace analyses of ilities that showed up on the Pareto Frontier for a prior set of ilities. This would allow architecture performance to be shown across graphs in a clear way.
- Updating the model to be n-dimensional with respect to mapping to ilities, potentially giving insight into clusters of architectures that are optimized for groups of ilities. Project inputs

could then be used as input into this model and could yield recommendations regarding the ideal architectures.

- Researching a way to model relationships between architectural decisions and the impact they have on project life-cycle properties. This could give a more complete picture of the expected results of different project architectures.
- Conducting more in depth analysis of the sensitivity of project architectures to specific architectural decisions.
- Testing of newly discovered project architectures in real-world settings to see how the teams perform and what additional insights can be gained from this process.
- Automating the tools and processes used in this research to highlight and cluster the most impactful architectural decisions on the performance on certain ilities.

10. References

[1] O. L. de Weck, "When is Complex Too Complex," at *Conference on Systems Thinking for Contemporary Challenges*, MIT, Cambridge, MA, Nov. 7, 2016. Available: https://www.youtube.com/watch?v=jvQkmR1XcyY

[2] B. Kitchenham *et al.*, "DESMET: A method for evaluating software engineering methods and tools," in *Computing & Control Engineering Journal*, vol. 8, no. 3, pp. 120-126, Jun. 1997.

[3] K. Inada, "Analysis of Japanese Software Business," M.S. thesis, System Design and Management Program, MIT, Cambridge, MA, 2010.

[4] T. Mitsuyuki *et al.*, "Evaluation of Project Architecture Mixing Waterfall and Agile by Using Process Simulation in Software Systems Project," in *Journal of Industrial Integration and Management*, vol. 2, no. 2, June 2017.

[5] A. M. Davis *et al.*, "A Strategy for Comparing Alternative Software Development Life Cycle Models," in *IEEE Trans. Softw. Eng.*, vol. 14, no. 10, pp. 1453-1461, Oct. 1988.

[6] R. Sorenson, "A Comparison of Software Development Methodologies," for *Softw. Technology Support Center*, United States Air Force, Hill AF Base, Ogden, Utah, 1995. Available: http://www.stsc.hill.af.mil/crosstalk/1995/01/Comparis.asp

[7] W. W. Royce, "Managing the Development of Large Software Systems," in *Proc., IEEE WESCON*, Aug. 1970, pp. 1-9.

[8] E. M. Simão, "Comparison of Software Development Methodologies based on the SWEBOK," Ph.D. dissertation, Dept. of Informatics, Universidade do Minho, Braga, Portugal, 2011.

[9] "Guide to the Software Engineering Body of Knowledge," *IEEE Computer Society*, P. Bourque and R. E. Fairley, Eds., 2014.

[10] R. Shaydulin and J. Sybrandt, "To Agile, or not to Agile: A Comparison of Software Development Methodologies," Available: https://arxiv.org/abs/1704.07469, 2017.

[11] N. M. A. Munassar and A. Govardhan, "A Comparison Between Five Models Of Software Engineering," in *International Journal of Computer Science*, vol. 7, no. 5, pp. 94-101, Sep. 2010.

[12] *Military Standard: Defense System Software Development*, 2167A, Feb. 1988.

[13] K. Beck *et al.* (2001). *Manifesto for Agile Software Development* [Online]. Available: http://agilemanifesto.org

[14] B. W. Boehm, "A Spiral Model of Software Development and Enhancement," in *Computer,* vol. 21, no. 5, pp. 61-72, May 1988.

[15] D. K. Sobek II *et al.*, "Toyota's Principles of Set-Based Concurrent Engineering," *MIT Sloan Manage. Review*, Winter 1999, Jan. 1999 [Online]. Available: https://sloanreview.mit.edu/article/toyotas-principles-of-setbased-concurrent-engineering/

[16] R. Ammar *et al.*, "Architectural design of complex systems using set-based concurrent engineering," in *2017 IEEE International Symp. on Systems Engineering (ISSE)*, Vienna, Austria, 2017, pp. 1-7.

[17] *Set-Based Design* (Apr. 11, 2018) [Online]. Available: https://www.scaledagileframework.com/set-based-design/

[18] E. Crawley *et al.*, *System Architecture: Strategy and Product Development for Complex Systems,* Hoboken, NJ: Pearson Higher Education, 2016, pp. 197-200.

[19] O. L. de Weck *et al.*, *Engineering Systems,* Cambridge, MA: MIT Press, 2011, pp. 65-96.

[20] B. R. Moser and R. T. Wood, "Complex Engineering Programs as Sociotechnical Systems," in *Concurrent Engineering in the 21st Century: Foundations, Developments, and Challenges*, J. Stjepandić *et al.*, Eds. Cham, Switzerland: Springer, 2015, pp. 51-65.

[21] "Ubuntu version history," in *Wikipedia* (May 9, 2019) [Online]. Available: https://en.wikipedia.org/wiki/Ubuntu_version_history

[22] "Mission statement," Blizzard Entertainment [Online]. Available: http://us.blizzard.com/en-us/company/about/mission.html

[23] S. Hannapel and N. Vlahopoulos, "Implementation of set-based design in multidisciplinary design optimization," in *Structural and Multidisciplinary Optimization*, vol. 50, no. 1, pp. 101-112, Feb. 2014.

[24] S. Hannapel *et al.*, "Including Principles of Set-Based Design in Multidisciplinary Design Optimization," in *12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference*, Indianapolis, IN, Sept. 2012, pp. 1-13.

[25] R. Miller et al., "Reading 10: More Efficiency," in *MIT User Interface Design & Implementation* [Online]. Available: http://web.mit.edu/6.813/www/sp18/classes/10-more-efficiency/

[26] J. Thomas, STAMP/STPA Lecture at MIT for System Design and Management Program, Spring, 2017.

[27] Lightning Kite, aggregated data from sampling of projects, Jul. 2015.

[28] *List of software development philosophies* (Jan. 6, 2018) [Online]. Available: https://en.wikipedia.org/wiki/List_of_software_development_philosophies

[29] B. Clark et al., "DoD Software Factbook," CMU, Pittsburgh, PA, 2015.

[30] "Aristotle," in *Wikipedia* (April 21, 2018) [Online]. Available: https://en.wikiquote.org/wiki/Aristotle

[31] A. W. Gray, "Enhancement of Set-Based Design Practices Via Introduction of Uncertainty Through the Use of Interval Type-2 Modeling and General Type-2 Fuzzy Logic Agent Based Methods," Ph.D. dissertation, Naval Arch. and Marine Eng., Univ. of Michigan, Ann Arbor, MI, 2011.

[32] D. Raudberget, "Pratical Applications of Set-Based Concurrent Engineering in Industry," in *Journal of Mechanical Engineering*, vol. 56, no. 11, pp. 685-695, 2010.

[33] D. Raudberget, "The Decision Process in Set-Based Concurrent Engineering– An Industrial Case Study," in *International Design Conference – Design 2010*, Dubrovnik, Croatia, 2010, pp. 937-946.

Appendix A: Justifications for Architectural Decision to Ility Scores

The following tables attempt to provide the justification for the scores for each architectural decision to ility mapping. This is intended to give insight into why the scores show up the way they do. Ideally, this can give insight into why a given project architecture could look different than expected based on a certain ility.

Ility Justifications: Responsiveness to Change

Decisions	Options	Score	Justification
Requirements Set (Scope)	Fluid	8	Fluid requirements allow teams to adjust respond to change.
	Fixed	1	Fixed requirements make it very difficult to respond to change.
	Ranges, w/ inc specificity	5	Having ranges that define over time allow for responding to change, but may not be entirely open-ended, not allowing complete responsiveness to change.
Unit of Work	User story	5	Easier to see what needs to be changed based on what's being delivered.
	Task	2	More difficult to update sometimes non-obvious tasks that result from upstream change.
Work Segmentation	End to End	5	Straightforward to change. Better encapsulation of functionality.
	Phased	1	Work spread across multiple phases makes change later in process more difficult.
Release Organization	Time Bound	8	If a delivery needs to happen by a certain date and a change is made, then whatever is not ready by the date simply doesn't ship. This makes for adapting to change very straightforward.
	Feature Bound	8	If a project is feature bound, then when the feature set is ready the product can ship. If there are changes to scope, this is also straightforward.
	Cost Bound	3	If a budget is given and a team is directed to go as far as they can on the budget provided, there is still a good amount of flexibility. However, because cost is a direct result of the work being executed (and therefore one level abstracted from the feature set) it can be less clear what the impacts of change will be on cost.
	Time/Feature Bound	2	With two constraints to manage, a project is less able to adapt.
	Time/Feat/ Cost Bound	1	Working with all three constraints of the iron triangle fixed is difficult to manage on its own. Introducing change makes this even more difficult.
Design Set + Concept Selection	SD + Solution Seeking	5	Working with only one solution makes this more adaptible in the event of changes.
	MD + Non-SS	3	When changes come in, each of the designs in the set have to absorb the change. However, it is possible that one of the designs already accounts for the change.
	Phased	5	Exploration happens in each of the phases of the project: design, coding, testing, etc. This provides more opportunities for exploration to take place.

Decisions	Options	Score	Justification
Exploration	Continuous	8	Exploration happening continually throughout a project, for example, in each interval if used in an architecture that supports this.
	Initial	3	Expoloration done only at the beginning of the project, perhaps a prototype.
	Not Executed	1	No specific exploration done.
Project Feasibility (When determined)	Inception	2	Feasibility study conducted at the outset of the project.
	Post-prototype	3	Feasibility determined after first creating a prototype.
	Not Done	1	No feasibility study conducted.
	Continuous	8	Feasbility conducted throughout the project, perhaps at each interval if project architecture supports this.
Budget (How managed)	Full Project Funding	1	If project is fully funded with a fixed budget set at the beginning, it's much harder to adapt each time a change comes in.
	Incremental Funding	8	If funding is allocated periodically or at increments throughout the project, adapting to change is not bound by a fixed budget.
	Perf-based Funding	5	Responsive to change based on performance. Ability to change funding to meet progress.
Quality Recovery (How managed)	Continual prioritization	8	Continually prioritizing rework allows for changes to be worked into project work.
	Post production	1	Applying rework at the end of production.
	Phased	2	Responsive to change at set points in the process.
Team Organization	Cross-Functional	3	Cross-functional teams are better able to quickly identify impacts of changes due to wide range of perspectives of team members.
	Functional	2	Certain changes could be handled.

Ility Justifications: Scope Scalability

Decisions	Options	Score	Justification
Requirements Set (Scope)	Fluid	1	Fluid requirement sets make scaling up to large teams and large projects difficult because there are more things to manage.
	Fixed	8	A fixed set of requirements reduces variability and therefore scales with less effort.
	Ranges, w/ inc specificity	3	While not entirely fixed, with set ranges the scope of what can change is reduced and therefore is able to scale moderately well.
Unit of Work	User story	3	Distributed volunteering for tasks could be more scalable, but harder to manage at large scale as well.
	Task	5	Could be better at scale because of centralized control, but harder for single body to understand the entire system.

Decisions	Options	Score	Justification
Work Segmentation	End to End	2	Does not scale extremely well due to intricate dependency webs of cross-functional individuals working on the same parts of the system for the end-to-end feature.
	Phased	5	Phased approaches scale because teams can focus on a set of tasks for a project and then send them off to the next phase/team when they're completed, reducing the amount of context switching.
Release Organization	Time Bound	3	Only a single variable constraint to manage, reduces complexity.
	Feature Bound	3	Only a single variable constraint to manage, reduces complexity.
	Cost Bound	2	As a derived value, this can be difficult to track as it's commonly learned after the development has taken place.
	Time/Feature Bound	2	Trackable variables to manage, but two dimensions add complexity.
	Time/Feat/ Cost Bound	5	While this approach may pose risks in other areas, it does scale since it attempts to limit variablitiy in all areas.
Design Set + Concept Selection	SD + Solution Seeking	5	Maintining a single design is easier to scale than maintaining multiple.
	MD + Non-SS	2	Given a large scope, maintaining multiple designs could be a lot more to manage.
Exploration	Phased	3	Phasing the exploration work introduces some opposition to a scope under scale as it introduces work at various times throughout the project.Â
	Continuous	2	Executing continuous exploration work introduces opposition to a scope under scale as it introduces work continuously throughout the project.
	Initial	5	Executing only initial exploration work scales as it contains all of the exploration work into a single segment.
	Not Executed	8	Not executing any exploration work scales as it does not add to scope, regardless of size.
Project Feasibility (When determined)	Inception	5	Product feasibility determination executed at inception confines the feasibility work to a specific segment of the project, not adding substantial work to the scope and scales well.
	Post-prototype	5	Product feasibility determination executed after a prototype confines the feasibility work to a specific segment of the project, not adding substantial work to the scope and scales well.
	Not Done	8	No product feasibility determination executed at all does not add work to the scope and scales well.
	Continuous	1	Continual product feasibility determination can add substantial work to the scope and could cause friction when applied at larger scales.
Budget (How managed)	Full Project Funding	5	Where projects are fully funded, teams have less risk of starts and stops throughout the duration of the project, making this approach, when possible, more scalable.
	Incremental Funding	3	When funding is provided based on some interval, this can be scalable, though consideration must be given to burn rates so the project budget spending does not outpace the allocation of funds. This can be more difficult in a large scale environment.

Decisions	Options	Score	Justification
	Perf-based Funding	2	When funding is based upon satisfaction of milestones or gates, this can indicate uncertainty for teams or limited funding when approaching a milestone. This can be complicated at large scale.
Quality Recovery (How managed)	Continual prioritization	2	When quality recovery (or rework) is continually prioritized, this can introduce substantial variability into the work execution, with many more work units to manage at high scale.
	Post production	5	When quality recovery is addressed at the end of the project, this can delay the need to reprioritize mid-stream, allowing teams to avoid redirecting their trajectory.
	Phased	3	When quality recovery occurs within each phase, the amount of churn that occurs due to rework can be minimized, reducing the potential for variability.
Team Organization	Cross-Functional	2	Cross-functional teams may not have a large number of individuals with the same skillset on the same project. New team members need to come up to speed both on the project as well as the skillset in order to adapt, causing difficulty in a high scale situation.
	Functional	8	A functional team, where team members all have very common skillsets, are able to adjust to scale and replace team members as needed, generally, more quickly.

Ility Justifications: Ease of Customer Collaboration

Decisions	Options	Score	Justification
Requirements Set (Scope)	Fluid	8	WIth a fluid requirement set, a project is able to respond quickly to customer feedback.
	Fixed	2	With a fix requirement set, a project is not quickly adaptable to customer feedback.
	Ranges, w/ inc specificity	5	With a requirement set based on ranges that increase over time in specifity, customers are able to give feedback and influence the direction within the scope of the ranges of the requirements.
Unit of Work	User story	8	With a user story, an end-to-end testable and understandable requirement, a customer is able to visualize and set when the full feature is functional, making providing feedback a more natural process.
	Task	2	Since tasks are more engineering-focused or focused on what needs to be done than the experience to create, these are not as condusive to customer interaction and feedback.
Work Segmentation	End to End	8	By segmenting the work as end-to-end, this creates a close link between the work being done and the experience it enables, allowing for more timely review and engagement from a customer.
	Phased	2	With work being executed in phases, the actually, fully functional feature is not generally usable and verifiable by a customer until later phases, collection customer feedback late in the development process.
Release Organization	Time Bound	3	Gives customer ability to say when project ships.
	Feature Bound	5	Feature set says when customer is ready for the product to launch.
	Cost Bound	3	Gives customer opportunity to say when to pull plug or add funds.

Decisions	Options	Score	Justification
	Time/Feature Bound	1	Hard to respond to customer feedback with mutliple variables binding project.
	Time/Feat/ Cost Bound	1	Hard to respond to customer feedback with mutliple variables binding project.
Design Set + Concept Selection	SD + Solution Seeking	3	Only one solution to choose from, but still opportunities to give feedback.
	MD + Non-SS	8	It's hard for customers to visualize something better than what they're seeing. Given alternatives in a multiple design set, they help choose which they like best.
Exploration	Phased	5	Phased more difficult to adapt to customer feedback.
	Continuous	8	Continuous offers many opportunities to adapt to customer feedback.
	Initial	3	Initial only allows for customer collaboration at beginnig.
	Not Executed	1	Unable to adapt to customer feedback if no exploration done.
Project Feasibility (When determined)	Inception	2	Only adaptable to customer feedback at the begnining.
	Post-prototype	3	Only allows adaptation through prototype.
	Not Done	1	Does not adapt to customer feedback.
	Continuous	8	Allows for customer colloboration throughout project.
Budget (How managed)	Full Project Funding	1	Unable to adapt once project kicks off.
	Incremental Funding	3	Moderately able to respond to customer feedback.
	Perf-based Funding	5	Better able to respond to customer feedback and collaboration.
Quality Recovery (How managed)	Continual prioritization	5	Better able to consider customer feedback as rework is done throughout.
	Post production	2	Less able to respond to customer feedback as rework is at the end.
	Phased	3	Hard to respond to customer feedback when only the later phases are demonstrable to customer.
Team Organization	Cross-Functional	5	Teams able to adapt to customer collaboration.
	Functional	2	Less able to adapt to customer feedback as teams are disjointed and siloed.

Ility Justifications: Ability to Identify Project Risk

Decisions	Options	Score	Justification
	Fluid	1	Changing requirements indicate changing risks.

Decisions	Options	Score	Justification
Requirements Set (Scope)	Fixed	5	Fixed set allow for more stable risks.
	Ranges, w/ inc specificity	8	Allows for more risk exploration due to more options.
Unit of Work	User story	8	Easier to see if feature will achieve need when realized end to end.
	Task	5	Easier to identify risk because required beforehand dissection of problems to be able to plan each individual task.
Work Segmentation	End to End	5	Easier to see if feature will achieve need when realized end to end.
	Phased	2	Phased has harder time seeing project risk because phases don't align to the way risks are realized.
Release Organization	Time Bound	2	Does not require as much forward looking, only identify risks to the next release.
	Feature Bound	3	Requires identifying ahead where features will be headed.
	Cost Bound	2	Does not require as much forward looking, only identify risks to the next release.
	Time/Feature Bound	5	Requires looking ahead at least on two dimensions of where product will go.
	Time/Feat/ Cost Bound	8	Requires looking ahead and identifying where the product will go.
Design Set + Concept Selection	SD + Solution Seeking	2	Only allows identification of risks based on a single design. Could be missing things.
	MD + Non-SS	8	Exploration of risks using multiple designs. Better able to identify potential risks as more designs are explored.
Exploration	Phased	3	While exploration not happening as frequently, does require more looking ahead.
	Continuous	5	Continuously exploring, however, not forced to look ahead.
	Initial	2	Looking ahead at risks at the beginning. But unable to do so during the project.
	Not Executed	1	No looking ahead at risks.
Project Feasibility (When determined)	Inception	2	Forces looking ahead at beginning but not throughout as new risks arise.
	Post-prototype	3	Forces looking ahead early on in the project, but not throughout as new risks arise.
	Not Done	1	Does not for looking ahead at risks.
	Continuous	5	Continously testing, but does not fors look ahead.
Budget (How managed)	Full Project Funding	2	Forces look ahead from the beginning, but does not force looking ahead throughout.
	Incremental Funding	3	Forces looking ahead at periods throughout the project.
	Perf-based Funding	5	Forces an assessment of risks upcoming each time new funding is decided to be allocated.

Decisions	Options	Score	Justification
Quality Recovery (How managed)	Continual prioritization	5	Offers oppotunities to relook at risks throughout.
	Post production	1	Pushes assessment of risks to end.
	Phased	3	Encourages looking at risks at points during the development.
Team Organization	Cross-Functional	5	Risks can be identified from differing perspectives.
	Functional	2	Risks identification siloed by team.

Ility Justifications: Ability to Meet Deadline

Decisions	Options	Score	Justification
Requirements Set (Scope)	Fluid	1	Doesn't force considering possibilities early on. Adaptive, but doesn't adapt well to hitting a deadline.
	Fixed	5	Able to hit a deadline because everything is planned out, but if change happens, can derail whole project.
	Ranges, w/ inc specificity	8	Good because some rigidity and forcing to look ahead, but still leaves room for adaptation as project evolves.
Unit of Work	User story	3	Requires looking at project as collections of features to satisfy user needs. No major impacts to hitting deadlines though.
	Task	5	Requires looking at project from top-down perspective which gives single group of individuals ability to set path to hitting deadline.
Work Segmentation	End to End	1	No particular direct impact on ability to hit deadline.
	Phased	1	No particular direct impact on ability to hit deadline.
Release Organization	Time Bound	3	Helps ensure something will be ready on a date, no guarantees how much of the scope though.
	Feature Bound	2	Focuses on getting the right set of features out, no guarantees on date though.
	Cost Bound	1	No focus on getting the right set of features out on a date and limits adding funding.
	Time/Feature Bound	8	Focus on feature set and date, leaves opening for adding more funding.
	Time/Feat/ Cost Bound	2	Limits ability to add more resources as need in order to hit timeline.
Design Set + Concept Selection	SD + Solution Seeking	2	All eggs in one basket.
	MD + Non-SS	8	Diversify options for hitting a deadline.
Exploration	Phased	3	Helps identify unknowns in stages.
	Continuous	5	Helps identify unknowns throughout project.

Decisions	Options	Score	Justification
Exploration	Initial	2	Helps identify unknowns early, but not throughout.
	Not Executed	1	Doesn't help identify unknowns, could hit major roadblocks.
Project Feasibility (When determined)	Inception	2	Feasibility tests done early on increase chances of hitting deadline, but not as things change throughout project.
	Post-prototype	3	Requires a prototype, helps to identify feasibility through working product.
	Not Done	1	Doesn't help determine feasibility, making harder to hit deadline.
	Continuous	5	Feasibility tests conducted throughout the life of project helps avoid steps that could not ultimately end up working and cause setbacks.
Budget (How managed)	Full Project Funding	3	Does not place accountability on hitting the individual milestones. But could also remove roadblocks for the right teams.
	Incremental Funding	2	Could cause delays or unnecessary bottlenecks due to funding allocations that don't match up with resources needed to hit deadlines.
	Perf-based Funding	5	Places motiviation on hitting individual milestones leading toward hitting deadline.
Quality Recovery (How managed)	Continual prioritization	3	Allows for reprioritization of tasks when rework is needed, but does not protect against slippage.
	Post production	3	Allows the progress to continue toward final delivery, but doesn't protect against delivering the wrong product.
	Phased	2	Allows for periodic rework, but can be costly if project has to return to prior phase.
Team Organization	Cross-Functional	1	No specific impact on ability to meet deadline.
	Functional	1	No specific impact on ability to meet deadline.

Ility Justifications: Progress Trackability

Decisions	Options	Score	Justification
Requirements Set (Scope)	Fluid	1	Hard to identify how much progress has been achieved.
	Fixed	8	Easier to track progress on fixed set of requirements.
	Ranges, w/ inc specificity	5	Somewhat fixed gives ability to track progress but not entirely.
Unit of Work	User story	5	Allows tracking progress by feature or user needs satisfied, not just by tasks.
	Task	5	Allows tracking from a top-down perspective.
Work Segmentation	End to End	3	Allows tracking across features.
	Phased	3	Allows tracking across phases.
	Time Bound	1	Does not force any specific tracking on scope.

Decisions	Options	Score	Justification
Release Organization	Feature Bound	5	Requires focus on scope and how far along it is.
	Cost Bound	1	Does not force any specific tracking on scope.
	Time/Feature Bound	3	Requires focus on scope and how far along it is.
	Time/Feat/ Cost Bound	3	Requires focus on scope and how far along it is.
Design Set + Concept Selection	SD + Solution Seeking	5	More straightforward way to track scope.
	MD + Non-SS	2	Difficult to track how far along the project is due to multiple concurrent designs.
Exploration	Phased	2	Impact to scope at various points in the project, making progress tracking more volatile.
	Continuous	1	Can be disruptive to scope making more difficult to track progress throughout.
	Initial	3	Minimizes impact to scope due to exploration
	Not Executed	5	No exploration executed doesn't cloud the scope (may have other side effects though)
Project Feasibility (When determined)	Inception	3	Limits impact to scope to early on, making scope more trackable throughout.
	Post-prototype	3	Minmizes impact to ccope since only done based on prototype created.
	Not Done	5	Minmizes impact to ccope since no feasibility testing is done and no changes need to be made.
	Continuous	1	Can have impact on scope throughout project, causing changes and making scope more volatile.
Budget (How managed)	Full Project Funding	5	WIth funding set, able to track scope progress throughout.
	Incremental Funding	3	Minimal impact to changing scope, minimal impact to trackability of scope.
	Perf-based Funding	2	Could be disruptive to scope throughout project development process.
Quality Recovery (How managed)	Continual prioritization	1	Could be very disruptive to scope causing for diminshed ability to track.
	Post production	2	Could be more trackable throughout project, but end could introduce substantial changes to scope.
	Phased	3	Impact to scope at various points in the project, making progress tracking more volatile.
Team Organization	Cross-Functional	1	Minimal impact to changing scope, minimal impact to trackability of scope.
	Functional	1	Minimal impact to changing scope, minimal impact to trackability of scope.

Appendix A: Python Scripts for Tradespace Generation

The following python scripts were used to import the architectural decision to ility mapping, which was maintained as a CSV file. The scripts were executed as notebooks in Jupyter, but could also be executed as standalone python scripts. The *tradespace_generatory.py* script reads in the mapping file, filters for the decisions that were enabled for a given execution, generates all possible combinations of the decisions, generates the Pareto Frontier, and exports the results to both CSV files and plotly for plotting on a scatter plot. The *pareto_interpretor.py* script uses the Paretor Frontier information created in *tradespace_generator.py* and counts by architectural decision. This provides information on dominated decisions along the Pareto Frontier. If a certain option has overwhelming presence over other options, it can be inferred that this decision has high impact on how the project architecture will perform with reference to the ilities in question. This also allows an individual to see clusters toward one ility or another.

tradespace_generator.py

```
1.    import itertools
2.    import csv
3.
4.    import plotly.plotly as py
5.    import plotly.graph_objs as go
6.    import numpy as np
7.
8.
9.    """
10.   ********************
11.   Config Parameters
12.   ********************
13.   """
14.   ILITY_1 = 0
15.   ILITY_2 = 1
16.   INPUT = 'input/ility_mappings.csv'
17.
18.   # Methodology Definitions
19.   METHODS = (
20.       #Agile
21.       {'1': 0, '2': 0, '3': 0, '4': 0, '6': 0, '7': 1,'10': 1,
22.           '11': 3, '17': 1, '18': 0, '29': 0},
23.       #Spiral
24.       {'1': 2, '2': 0, '4': 0, '6': 1, '7': 0, '10': 1,
```

```
            '11': 3, '18': 0},
        #Waterfall
        {'1': 1, '2': 0, '3': 1, '4': 1, '6': 1, '7': 0, '10': 2,
            '11': 0, '17': 0, '18': 1, '29': 1},
        #Setbased
        {'1': 2, '2': 1, '7': 1, '10': 1, '11': 2, '18': 0}
    )
    """
    ********************
    End Config Parameters
    ********************
    """

    """
    ********************
    Helper Methods
    ********************
    """
    def pareto_frontier(Xs, Ys, Ls, maxX = True, maxY = True):
        myList = sorted([[Xs[i], Ys[i], Ls[i]] for i in
        range(len(Xs))], reverse=maxX)
        p_front = [myList[0]]
        for pair in myList[1:]:
            if maxY:
                if pair[1] >= p_front[-1][1]:
                    p_front.append(pair)
            else:
                if pair[1] <= p_front[-1][1]:
                    p_front.append(pair)
        p_frontX = [pair[0] for pair in p_front]
        p_frontY = [pair[1] for pair in p_front]
        p_frontL = [pair[2] for pair in p_front]
        return p_frontX, p_frontY, p_frontL

    def trans_label(label, dec_ops):
        indiv = label.split('|')
        ops = []
        for i in indiv:
            dec_id, op_id = i.split(':')
            ops.append('%s) %s' % (dec_id, dec_ops[dec_id]
        [int(op_id)]))
        return '; '.join(ops)

    '''
    Convert the decision and relative option index "28:1"
    to the raw option index: 29
    '''
    def op_offset(dec_pair):
        dec_id, op_id = dec_pair.split(':')
```

```
72.         return dec_id, int(op_id)
73.     """
74.     ********************
75.     End Helper Methods
76.     ********************
77.     """
78.
79.     DEC_START_ROW = 6 #From 0
80.     Q_START_COL = 2 #From 0
81.     Q_DEF_ROW = 3 #From 0
82.     Q_INC_ROW = 2 #From 0
83.
84.     mappings = []
85.     dec_key = []
86.     include = []
87.     options = []
88.     il1_name = ''
89.     il2_name = ''
90.     with open(INPUT) as csvfile:
91.         reader = csv.reader(csvfile)
92.         for i, row in enumerate(reader):
93.             if i == DEC_START_ROW+ILITY_1:
94.                 il1_name = row[0]
95.             if i == DEC_START_ROW+ILITY_2:
96.                 il2_name = row[0]
97.             if i == Q_DEF_ROW:
98.                 dec_key = row[Q_START_COL:]
99.             if i >= DEC_START_ROW:
100.                mappings.append(row[Q_START_COL:])
101.            if i == Q_INC_ROW:
102.                include = row[Q_START_COL:]
103.            if i == DEC_START_ROW-1:
104.                options = row[Q_START_COL:]
105.
106.    current = None
107.    start = 0
108.    decisions = {}
109.    print(include)
110.    for i, dec in enumerate(dec_key):
111.        if dec != '':
112.            if current is not None and 'Yes' in include[start:i]:
113.                decisions[current] = (start, i)
114.            start = i
115.            current = dec
116.    if current is not None and 'Yes' in include[start:i]:
117.        decisions[current] = (start, len(dec_key))
118.
119.    method_min_max = []
120.    for m in METHODS:
```

```
121.        i1 = [0,0]
122.        i2 = [0,0]
123.        for d_id, offs in decisions.items():
124.            vals_1 = mappings[ILITY_1][offs[0]:offs[1]]
125.            vals_2 = mappings[ILITY_2][offs[0]:offs[1]]
126.            if d_id in m:
127.                i1[0] += int(vals_1[m[d_id]])
128.                i1[1] += int(vals_1[m[d_id]])
129.                i2[0] += int(vals_2[m[d_id]])
130.                i2[1] += int(vals_2[m[d_id]])
131.            else:
132.                i1[0] += int(min(vals_1))
133.                i1[1] += int(max(vals_1))
134.                i2[0] += int(min(vals_2))
135.                i2[1] += int(max(vals_2))
136.        method_min_max.append([i1, i2])
137.
138.    print(decisions)
139.    print('IL1: %s' % il1_name)
140.    print('IL2: %s' % il2_name)
141.    print(method_min_max)
142.
143.    ilities = [[],[],[]]
144.    dec_ops = {}
145.    for key, offs in decisions.items():
146.        ilities[0].append(mappings[ILITY_1][offs[0]:offs[1]])
147.        ilities[1].append(mappings[ILITY_2][offs[0]:offs[1]])
148.        l = len(mappings[ILITY_1][offs[0]:offs[1]])
149.        dec_ops[key] = options[offs[0]:offs[1]]
150.        ilities[2].append(['%s:%s' % (d, o) for d, o in zip([key]*l,
           range(0,l))])
151.
152.    il_1 = list(itertools.product(*ilities[0]))
153.    il_2 = list(itertools.product(*ilities[1]))
154.    il_3 = list(itertools.product(*ilities[2]))
155.
156.    x = []
157.    y = []
158.    labels = []
159.    for i, el in enumerate(il_1):
160.        x.append(sum(map(int, il_1[i])))
161.        y.append(sum(map(int, il_2[i])))
162.        labels.append('|'.join(il_3[i]))
163.
164.    x_np = np.asarray(x)
165.    y_np = np.asarray(y)
166.    l_np = np.asarray(labels)
167.
```

```
168.  p_front = pareto_frontier(x_np, y_np, l_np, maxX = True, maxY =
          True)
169.
170.  trace = go.Scatter(
171.      x = x_np,
172.      y = y_np,
173.      mode = 'markers',
174.      text = labels
175.  )
176.
177.  trace2 = go.Scatter(
178.      x = p_front[0],
179.      y = p_front[1],
180.      mode = 'lines+markers',
181.      text = p_front[2]
182.  )
183.
184.  data = [trace, trace2]
185.
186.  py.iplot(data, filename='%s-vs-%s' % (il1_name.lower().replace('
          ', '-'),
187.                                        il2_name.lower().replace('
          ', '-')))
188.
189.  with open('output/ility_output.csv', 'w+') as csvfile:
190.      writer = csv.writer(csvfile)
191.      for i, el in enumerate(il_1):
192.          writer.writerow(['|'.join(il_3[i]), sum(map(int,
          il_1[i])), sum(map(int, il_2[i]))])
193.
194.  with open('output/pareto_output.csv', 'w+') as csvfile:
195.      writer = csv.writer(csvfile)
196.      for i, el in enumerate(p_front[0]):
197.          writer.writerow([p_front[2][i], p_front[0][i], p_front[1]
          [i]]+trans_label(p_front[2][i], dec_ops).split('; '))
```

pareto_interpreter.py

```
1.    import pandas as pd
2.
3.    dmap = {}
4.    ops = []
5.    cols = ['il1', 'il2']
6.    for k, v in decisions.items():
7.        dmap[k] = len(cols) + len(ops)
8.        for i in range(*v):
9.            ops.append('d%s-%d' % (k, i))
10.   cols += ops
```

```
11.
12.    data = []
13.    for i, v in enumerate(p_front[0]):
14.        fresh = [0]*len(cols)
15.        fresh[0] = p_front[0][i]
16.        fresh[1] = p_front[1][i]
17.        for p in p_front[2][i].split('|'):
18.            dec_id, offset = op_offset(p)
19.            fresh[dmap[dec_id]+offset] = 1
20.        data.append(fresh)
21.
22.    df = pd.DataFrame(data, columns=cols)
23.    df.groupby(['il1', 'il2']).sum().to_csv('output/
           computed_decisions.csv')
24.    df.groupby(['il1', 'il2']).sum()
```