



**Institute for Software Research**  
University of California, Irvine

## Applying Software Design and Requirements Engineering Techniques to System Conception



Leyna C. Cotran  
University of California, Irvine  
leynacotran@gmail.com



Richard N. Taylor  
University of California, Irvine  
taylor@ics.uci.edu

March 2010

ISR Technical Report # UCI-ISR-10-1

Institute for Software Research  
ICS2 221  
University of California, Irvine  
Irvine, CA 92697-3455  
[www.isr.uci.edu](http://www.isr.uci.edu)

[www.isr.uci.edu/tech-reports.html](http://www.isr.uci.edu/tech-reports.html)

# Applying Software Design and Requirements Engineering Techniques to System Conception

Leyna C. Cotran, Richard N. Taylor

Institute for Software Research  
University of California, Irvine  
Irvine, CA 92697-3425  
{lccotran, taylor}@ics.uci.edu

ISR Technical Report # UCI-ISR-10-1

March 2010

## **Abstract**

While classical software engineering dictums separate the development of a system's requirements from its design, practice has largely shown this to be either impractical or naive. Much of the design of large software-dependent systems comes from prior systems, and that knowledge affects the requirements for new systems. Even if prior systems do not directly determine new system's requirements, the activity of developing requirements is, in practice and emerging theory closely intertwined with the activity of designing the system's structure and correlating design decisions. This survey examines requirements engineering techniques, assessing them with regard to how well they support the inclusion of prior design knowledge and how well they support the co-development of designs. The survey is based on the application of an evaluation framework to a set of well-known requirements engineering techniques. The framework includes criteria such as mapping to architecture, expressibility of design choices, and accessibility to relevant information. The techniques surveyed include Problem Frames, Use Cases, and Agile. The evaluation results in identifying a set of outstanding holes to be fixed in Requirements Engineering.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Survey Scope</b>	<b>6</b>
2.1	Large-scale Systems . . . . .	6
2.2	What is not addressed in this Survey . . . . .	6
2.3	Terms and Definitions . . . . .	6
<b>3</b>	<b>Highlights of Requirements Engineering</b>	<b>8</b>
3.1	Requirements Engineering Process with respect to Waterfall and Spiral . . . . .	8
3.2	Requirements Engineering Perspectives . . . . .	10
3.2.1	Coordination Perspective . . . . .	10
3.2.2	Intertwinement Perspective . . . . .	11
3.2.3	Technical Perspective . . . . .	13
<b>4</b>	<b>Problem Analysis</b>	<b>14</b>
4.1	Dispersal of Critical Information . . . . .	14
4.2	Articulation of Concepts . . . . .	14
4.3	Analyzability of Requirements . . . . .	15
<b>5</b>	<b>Evaluation Framework</b>	<b>17</b>
5.1	Requirement Engineering Techniques . . . . .	17
5.1.1	Problem Frames . . . . .	17
5.1.2	Viewpoints . . . . .	18
5.1.3	Use Cases . . . . .	18
5.1.4	Formal Language . . . . .	18
5.1.5	Test-Driven Development . . . . .	18
5.1.6	Domain Modeling . . . . .	19
5.1.7	Business Modeling . . . . .	19
5.1.8	Goal-oriented Approaches . . . . .	19
5.1.9	Agile Development . . . . .	20
5.2	Evaluation Criteria . . . . .	20
<b>6</b>	<b>Evaluation</b>	<b>23</b>
6.1	Problem Frames . . . . .	23
6.2	Viewpoints . . . . .	25
6.3	Use Cases . . . . .	27
6.4	Formal Language . . . . .	29
6.5	Test-Driven Development (TDD) . . . . .	30
6.6	Domain Modeling . . . . .	32
6.7	Business Modeling . . . . .	33
6.8	Goal-Oriented Approaches . . . . .	34
6.9	Agile Development . . . . .	36
6.10	Ad-Hoc Approach . . . . .	38

<b>7</b>	<b>Summary Analysis</b>	<b>40</b>
7.1	Evaluation Framework Limitations . . . . .	42
<b>8</b>	<b>Survey Insights</b>	<b>43</b>
8.1	Future Work . . . . .	45
<b>9</b>	<b>References</b>	<b>46</b>

## List of Figures

1	Royce’s real Waterfall Model [81] . . . . .	8
2	The Spiral Model [14] . . . . .	9
3	Perspectives of Requirements Engineering . . . . .	10
4	The Twin Peaks Model [72] . . . . .	12
5	The Problem Frame [49] . . . . .	18
6	The Domain, The Machine, and Shared Characteristics[48] . . . . .	32
7	RETs tackle different software development problems . . . . .	44

# 1 Introduction

The techniques taken today to perform requirements engineering and analysis activities in complex systems have some problematic areas. These problematic areas are the separation of requirements tasks from the design, context is difficult to assess from requirement wording, domain-knowledge is not always enough, and aspects of the system are inconsistently captured by different experts in the system.

System design and requirements activities have historically been two very separate tasks in traditional software engineering dictums. The Waterfall[81] and Spiral[14] models, for instance, depict the separation of these two activities. However, practice has largely shown that the separation of design and requirements is unrealistic[5]. The Twin Peaks Model[72] depicts the more realistic relationship between a system's architecture its' requirements. The Twin Peaks model shows that that the design decisions and requirements of a system are tightly coupled[86, 78].

Several examples in practice show that capturing requirements revolves around capturing standalone statements in a document, with no context provided about what the requirements mean and what type of system is needed. In aerospace projects for example, requirements are captured as a structured set of "shall" statements in a document that follows a military standardized format of writing requirements[1]. "Shall" statements, without context of the systems design, form standalone requirements with many possible interpretations. The statements may be inadequately written, leaving developers to interpret what the system does and the rationale behind the system's design. Problems with verification occur when the textual requirements are written too abstractly or incorrectly, inhibiting their testability. The result is tests of standalone statements that do not correspond to the system's design.

The requirements, in order to be effective and meaningful, must co-evolve with the design decisions made about the system. For instance, changing functionality in code reveals more understanding of the system. Design decisions may change due to alterations during implementation of functionality. In turn, requirements may also be affected because of a code change. This evolution is a challenge as these changes ripple through the system and the requirements must keep up with these changes.

Capturing requirements is not a simple task; it involves significant domain knowledge in order to gain insight to possible requirements. For instance, the spacecrafts domain has requirements concerning communication with ground operations and in flight data downlink and uplink capabilities. These domain requirements further decompose into specific areas of ground commanding and telemetry. The categories of requirements in the spacecraft example are the resulting phenomena of the application domain[50] of a spacecraft. Jackson suggests that domains share similar characteristics and products of that domain also share similar requirements. This observation holds for systems designed from the engineering knowledge of previously built systems. Previous knowledge affects and can dictate the requirements for a new system. While domain knowledge is largely useful for capturing the requirements of a system, the reality is each system will have a number of unique requirements that may not share similar characteristics across the same domain.

Different aspects of a system may be captured across different experts in the project. Capturing requirements involves significant coordination with multiple disciplines and experts. Crucial information that affects the requirements of a system tends to be scattered throughout a project among these different disciplines and system experts. Each expert or discipline has a distinctive approach to capturing the information it deems crucial. For example, developers capture their information in code and requirements engineers capture their information in a number of documents or a repository. This can ultimately lead to mismatches in the information captured. For example, critical data found in code and code structure affect requirements that may need to be changed. A significant change in code structure can introduce new requirements that will need to be analyzed and assessed. This scattering of information, and its inconsistent capture among the different disciplines in a project, causes problems in capturing requirements correctly because reconciling these

differences can be difficult.

This survey offers three contributions. First, the survey examines requirements engineering techniques, assessing them with regard to how well they support the inclusion of prior design knowledge and how well they support the co-development of design. Second, the survey presents an evaluation framework and evaluates each requirements engineering technique against the framework. Finally, the survey offers recommendations for future research based on the outcome of the evaluation.

This survey is structured as follows: Section 2 describes this survey's scope, Section 3 provides a brief history and various perspectives of Requirements Engineering, Section 4 elaborates on the problem this survey addresses, Section 5 discusses the evaluation framework (matrixed in Appendix A), Section 6 presents the different approaches as seen in the light of the evaluation framework, and finally, Section 7 offers future recommendations for this research, and concludes this survey.

## 2 Survey Scope

This section describes the category of systems that are addressed and subtopics of Requirements Engineering that will not be addressed in this survey.

### 2.1 Large-scale Systems

This survey primarily focuses on large-scale complex systems, where software is a subsystem of a larger system. This category of systems has software that must interplay with other systems that are logically separated from the main software system. Large-scale systems have a significant dependency on their software for logic execution of events and actions the system will take. Examples include governmental and aerospace projects such as satellites and spacecrafts, a registrar management system for a university that manages students' personal information and grades, or complex medical devices such as an insulin pump. These systems are mission-critical, where a loss of data or error in functionality can have major consequences to the system's success.

There are several factors that make large-scale systems interesting to explore. First, there is a significant interplay between major elements of the system. Specifically, the design decisions made of the system and the requirements that are captured for the system have a strong dependency on one another. With large-scale system, the management of this interplay is a challenge because of the scale of the system. Designing a system and capturing requirements are hard enough as it is, but when the system's scale is much bigger, the difficulty of the relationship between design and requirements scales up as well.

Second, the number of people involved in developing systems makes this interplay even more interesting because the coordination involves different perspectives. These perspectives need to be resolved in order for the system to succeed in its development.

Third, the amount of time it takes to build large-scale systems spans on an order of years. The span of time it takes to develop large-scale systems (on the order of years) allows iterations of the system to occur and reoccur, where incremental improvement is made as the system evolves.

### 2.2 What is not addressed in this Survey

Requirements Engineering involves extensive human and social aspects particularly in areas of elicitation and negotiation between different parties that have a stake in the system. While much of what is discussed in this survey will indirectly address such concerns, it is not the intent of this survey to examine and analyze these social-centered processes and assess the merits behind people's actions. Topics such as ethnography, workflow process, and social science topics under Requirements Engineering will not be addressed (for work in these areas, see [37, 92, 45, 3, 76]).

Breakdown in organizational communication, people's political motivations in an organization, pressures to meet a scheduled delivery date, and business politics, for instance, can have a profound impact on the design decisions chosen for a system. This in turn also have impacts to the requirements of a system as well. However, it is assumed in this survey that if the world was perfect, if the communication among designers and requirement engineers was perfect, if other human-centered issues were also perfect, that the issues discussed in this survey with regard to design and requirements would still exist and perhaps make these problems even worse. The conclusions of this survey would still not change. In order to bound the problems discussed, it is assumed these social and human-centered issues exist and therefore this survey will focus on the problems of requirements engineering issues with respect to design choices made of the system.

### 2.3 Terms and Definitions

Five terms and definitions are defined here as these terms are used consistently throughout this survey.

**Requirements Engineering:** Requirements Engineering is the process taken to discovering the purpose of the system[73]. [73] further expands this definition to put software requirements engineering in a system’s context; in fact, characterizing Software Requirements Engineering as a branch of Systems Engineering because “software cannot function in isolation from the system in which it is embedded.”

**Architecture and Design:** For this survey, “Architecture” is defined per [86]: “A software system’s architecture is the set of principal design decisions made about the system.” [86] further describes two significant terms in this definition that are important to an architecture. First, *design decisions* includes all aspects of the system, such as structure, functionality, interdependencies, non-functional properties, and implementation. Second, the fact that these set of decisions are *principal* suggests the importance in granting a design decision as part of an architecture (an “architectural design decision”). Ultimately those who are invested in the product (architect included) will decide which design decisions are should be included in the system’s architecture. Moreover, the term *architecture* and *design* can be somewhat used interchangeably. For this survey, Architecture and Design are addressed distinctly in the survey’s evaluation. Architecture includes defining architectural styles, architectural description languages, components and connector structure and definition, and interfaces. Design includes addressing key decisions that influence the structure, implementation, and principal properties of the system.

**Traceability:** Traceability is the activity of tracing a written requirement bi-directionally[38]: from the requirement’s source to the requirement’s decomposition. For this survey, this definition is extended to include traceability to key pieces of information of the system’s design (contributing to the system’s architecture but may not necessarily be captured as a written requirement).

**Systems Engineering:** Systems Engineering is the process and activity associated with developing a system, of which software is an integrated element of that system. Depending on the system, the Systems Engineering discipline could include the software, hardware, firmware, electrical, and mechanical development of that system. Classical Systems Engineering development follows the “V” (or “Vee”) Model [35].

### 3 Highlights of Requirements Engineering

F. Brooks was one of the first to observe the inherent difficulties of capturing requirements[18]:

*“The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed requirements...No other part of the work cripples the resulting system if done wrong. No other part is as difficult to rectify later. Therefore, the most important function that the software builder performs for the client is the iterative extraction and refinement of the product requirements.”*

Historically, requirements development is often separated into its own task in the software and system development lifecycle. The Waterfall Model and the Spiral Model are models that are widely practiced in the software engineering discipline. Both models will be discussed here for purposes of setting the context of where the Requirements Engineering discipline has found itself over the last 40 years in software development.

#### 3.1 Requirements Engineering Process with respect to Waterfall and Spiral

Royce[81] noted the separation of building requirements from programming design in his illustration of the Waterfall Model, which has feverishly been taken out of context to become the primary method of developing software. In fact, what Royce tells his audience is that the successive steps for software and system development simply cannot work because errors are discovered later rather than sooner, and modification of the requirements and design is needed.

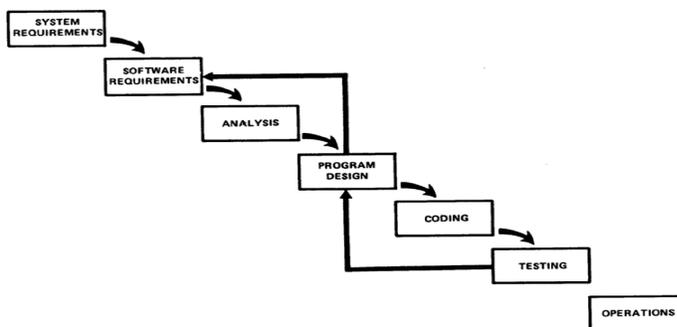


Figure 1: Royce’s real Waterfall Model [81]

Figure 1 depicts the crux of the requirements problem on large systems: system requirements need to be uncovered first in parallel with software requirements discovery. Often, testing the system reveals design issues that require reworking the software requirements. These problems affect the analysis and elicitation phases of development and force rework of the whole lifecycle.

Since the time Royce wrote about these interdependency issues in 1970, many advances have been made in tools, technologies, and development processes. In 1988, Boehm derived the Spiral Model[14], which addressed the issue of unknown upfront requirements, and allowed prototype iterations of the software system in order to “discover” the requirements and build a complete system.

In Figure 2, the Spiral Model demonstrates the incremental activities of “discovery” of the system through an early concept of operations, early prototyping, and requirements development. The concept behind the Spiral Model is that for every iteration (“spiral”) there are a number

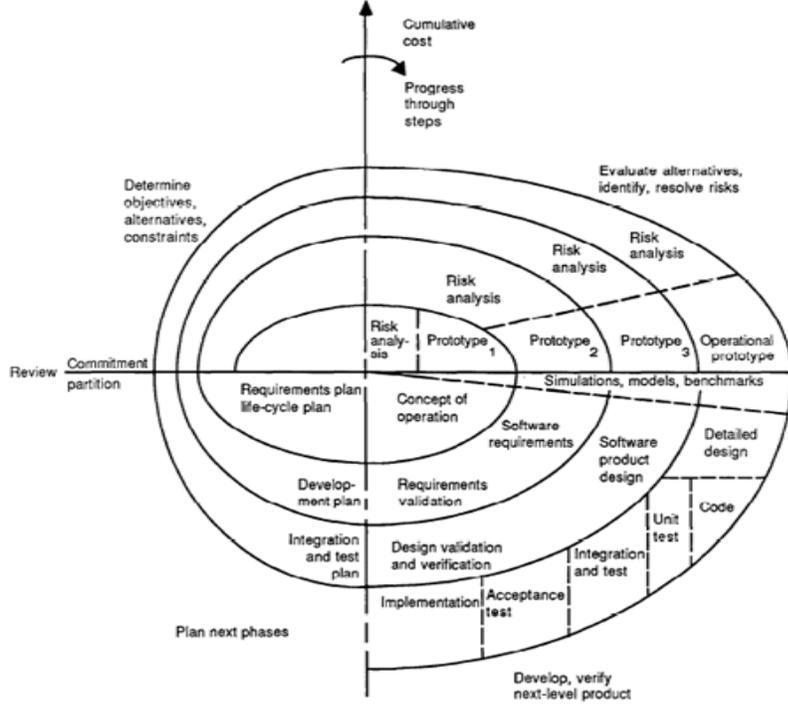


Figure 2: The Spiral Model [14]

of serially executed activities. Each “spiral” or iteration of these activities evolves with more information from the previous “spiral”, and this iteration continues a number of times until the system requirements are better understood. However, if we take a closer look at the Spiral Model, there remains a separation of requirements from product design (depicted in Figure 2 between Spiral # 2 and Spiral # 3), as it is assumed that after a short number of iterations, the requirements have been captured and design can begin.

The Spiral and Waterfall models both show the requirements phase is isolated from design activities. Royce’s Waterfall model shows the separation of the requirements phase is not practical because the need to re-assess requirements happens frequently. The Spiral model, despite its iterative nature, still calls for the separation of the requirements phase from the design phase. There are coordination, process, and technical aspects of both these models that are not entirely demonstrated.

Over the last twenty years, the Requirements Engineering discipline has fostered a number of sub-disciplines, namely in elicitation, negotiation, analysis, and documentation [99, 46, 76, 51]. These sub-disciplines revolve around coordination, process, and technical approaches. Requirements Engineers play multi-functional roles of managing the experts involved with designing the system (including their goals for the system), managing change control of the data throughout the system’s development, and striving for the completeness of the work being done on such large, complex systems[73].

## 3.2 Requirements Engineering Perspectives

For both systems development and software development, Requirements Engineering is a multi-disciplinary, interdisciplinary, and human-driven engineering activity. The requirements engineer must draw upon a variety of disciplines, utilize tools and techniques that can handle information change and evolution, and must, above all, master the skills of coordinating engineers from a variety of disciplines to capture a system’s requirements. The requirements engineer must organize captured information and adapt to the changes of the system. Often requirements engineers find that they have to become experts in specific engineering disciplines in order to capture relevant requirements.

The Requirements Engineering discipline can be further divided into three main perspectives: a coordination perspective, an intertwinement perspective, and a technical perspective (shown in Figure 3). These three perspectives are an abstraction of the work found in [73].

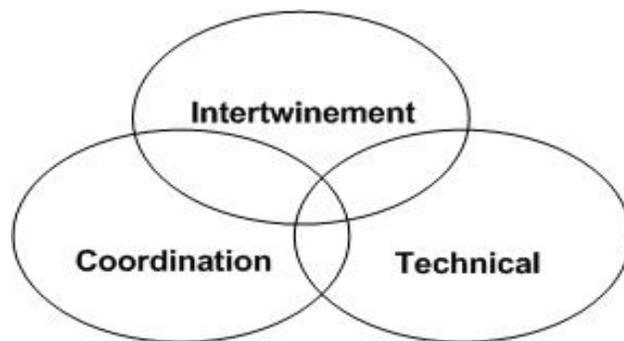


Figure 3: Perspectives of Requirements Engineering

### 3.2.1 Coordination Perspective

Understanding the requirements of the system often requires significant coordination with system experts in order to capture desired requirements. Moreover, projects that are developed by a globally distributed team (most aerospace and defense projects deal with this specific situation) must coordinate people and their ideas, work processes, and collaborative tools to capture shared information. Such tasks are required for efficient use of time to develop complex systems.

Software development is a collaborative activity, but there are several limiting factors when it comes to software development, including human limitations of organization, memory, historical information, and error identification. In [96], Whitehead explains that although collaborative tools and integrated development environments exist for software developers to work together, these tools exist for different phases of software development and collaborating artifacts and ideas co-exist over a variety of environments. This poses several problems. First, artifacts that are maintained in different phases of software development (architectural artifacts, requirement artifacts, and code, for instance) are built differently and involve different tools and outputs. Mapping information for the complete solution to the system’s design is inherently difficult because of the lack of consistency in these products. These products capture different yet pertinent information. Second, because different software experts build these products and speak different “languages” and terminologies, reconciling these differences can stunt progress if communication is not managed well. Building these products in different settings makes collaboration for complex systems very difficult, particularly in proceeding with good decisions about the design.

Coordination of informational artifacts poses several problems to understanding why decisions were made about a system, and understanding design solutions for the system. Several of these artifacts may lack history and are informally managed and out of date. Mapping design decisions to

such information can be difficult because that information can become volatile. In one case study[7], an extended traceability system was developed to tackle the Traceability Benefit Problem[6] by keeping track of coordinated information with metrics such as requirements creep, changes, progress, customer changes, and reusable information. This coordination extended to customer furnished documents, test cases, and specifications. While there were successes in mapping such information, this information was static: the traceability existed for historical data and could not keep up with the evolution of the information needed to capture requirements. There is significant risk of mapping data that is inaccurate and outdated when design decisions are not involved.

Collaboration techniques in software have addressed human limitations in developing systems and working with together. The output of these collaborative techniques is a number of formal and informal artifacts including specifications. The quality of these specifications is directly proportional to the conciseness of these artifacts produced. The pitfalls in coordination of vital system information often results in poorly written specifications.

Whitehead suggests for a broader participation in design from customers[96], capturing rationale behind decisions of architecture and design such that design choices will not be accidentally violated as new team members join the project, or the customer decides that they want to incorporate a new functionality that drifts from the intended design. Agile and Extreme Programming development methods, to some extent, address this by incorporating the customer in the actual software development with user stories, customer-written test cases, and gaining continuous feedback through coding and development[10]. This somewhat tackles the issue of gaining customer insight to implementation and coding, but requirements development is still an issue with Agile developed systems. With Agile development for instance, iterations are done in short spurts and customer desires (known as “features”) are priorities per iteration. Not all requirements might be captured during each iteration and these requirements run the risk of not being completed or ever implemented. While the coordination exists in Agile systems, the requirements work is either cursory or overlooked. A specification is generally never written in Agile and XP, and if so, the specification is often informal[75].

[79] looks at the Requirements Engineering discipline and re-orient us by looking at the Requirements Engineering discipline as a series of problems. These problems are specifically broken down to a number of sub-problems: a transformation problem (an evolution), a work piece problem (where a specification is taken as gospel), a informational display problem (inconsistencies in structured models), and as a behavioral (a process) problem. In fact, the trend is that requirements, when written, are factually correct and unchangeable. Such a view ties directly with [38] with the “pre-Requirements Specification” and “post-Requirements Specification” phases that address the lack of information known in the “pre-Requirements Specification” phase can lead to major traceability issues during requirements development. [79] attempts to put the people problem first in suggesting that Requirements Engineering encompasses a communication perspective that is clinically known to go awry because of natural flaws in human interaction.

### 3.2.2 Intertwinement Perspective

In earlier work[85], the intertwinement of requirements and design is discussed in regards to *specification* and *implementation*. Swartout and Balzer advocate that both should not be segregated. From our definition of “architecture”, it is noted that aspects of implementation may be part of the architectural design decisions of a system. [85] describes the current state of process approaches (circa 1982, when the Spiral model had not been published yet) where requirements development was a separate phase (and sequentially before) implementation.

Several other research classify this intertwinement as an iterative and incremental process [62, 14, 19] instead, where several iterations are taken to re-develop and evolve the system’s requirements and implementation. These iterative approaches are the most commonly used approaches for a system’s development. It is the intertwining of these phases that provides completeness in the work

involved in requirements and architectures.

In [72], Nuseibeh proposes the Twin Peaks model by describing the intertwined relationship between architecture and requirements development.

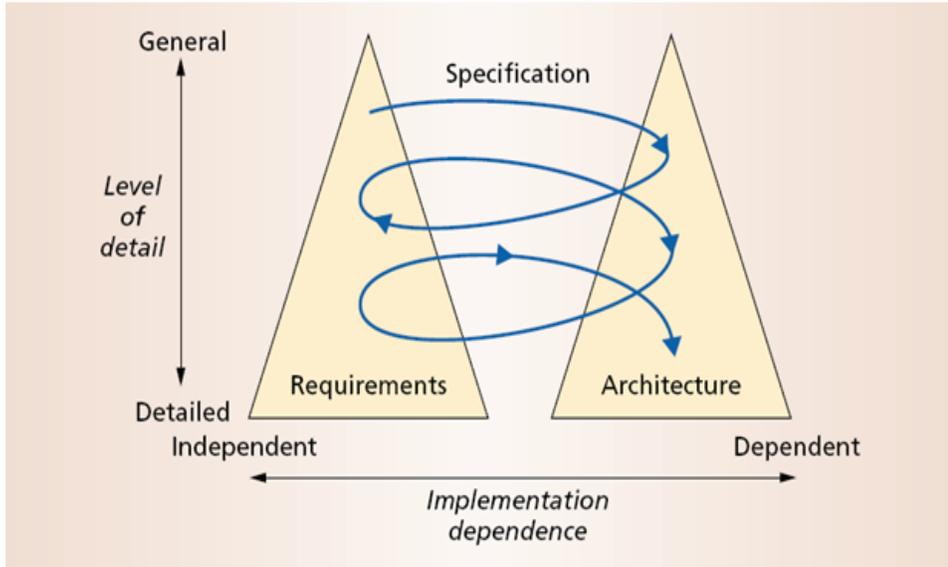


Figure 4: The Twin Peaks Model [72]

In Figure 4, the Twin Peaks Model emphasizes the co-evolution of requirements and architecture. Moreover, the Twin Peaks Model takes into account the level of detail of the information as well as the dependency on code and implementation. It is generally not recommended to add implementation details to the requirements specifications but to add goals and intents of the system as requirements. The Twin Peaks Model takes this into account on each axis (“Level of Detail” versus “Implementation Dependence”).

The Twin Peaks Model offers a development approach to software design development by suggesting that requirements analysis and architectural work happen in parallel. The Twin Peaks Model and [85], in essence, are suggesting there is a strong interdependence between requirements and design and that the approach to doing the work of requirements and design must be done in parallel. Requirements work in what is described in [85] and [72] is actually referring to the analysis needed on what is expected of the system. Documenting this work is merely one aspect, the solutions to meeting the goals and objectives of the system, its implementation, and the design rationale behind that solution fundamentally must be intertwined with the requirements.

An extension of the Twin Peaks Model can be found in later works with the CBSP (Component-Bus-System, and Properties) Approach[32], where the Twin Peaks Model is applied to the goal of bridging requirements and architectures through an intermediate approach. This work identifies a number of relevant relationships to relate a requirements engineering approach with an architectural approach. The CBSP approach provides an intermediate language for representing requirements in an architectural fashion.

There are other variations of the Twin Peaks Model, whether realized or not, that are commonly used in requirements work. XP methods[10] overall apply this parallel approach with the use of user stories, customer-written tests, and feature-estimation. In one case study[77], Agile methods were applied to the requirements development process for a multidisciplinary software project. Much intertwining of requirements work with customer needs and software development was done in order to address the challenges of having a non-technical customer and a distributed global team.

Approaches to requirements work involves process to keep the team organized. Many defense contracts require this formality in process in order to evaluate the work of the contractor. In [8], Aubrey describes in great detail the processes followed at General Dynamics, a defense contractor, in order to meet project milestones. Requirements and architecture work is a significant measure of progress to customers, particularly in design reviews where requirements and architecture are the prime topics of discussion for the formal milestones of System Readiness Reviews (SRRs), Preliminary Design Reviews (PDRs) and Critical Design Reviews (CDRs).

### 3.2.3 Technical Perspective

As previously discussed in this survey, the process and approach to doing requirements analysis must be decided and the coordination with other expertise in the system's development must exist. Another crucial perspective in requirements work is the technical perspective - involving the actual development of requirements products that are complete, sound, and meaningful to making decisions about the system's structure.

Modeling and analysis tools are useful in capturing desired behavior of the system while understanding the impacts of these behaviors. Understanding the information captured for large complex systems allows designers to proceed with better decisions of the system. Written requirements are not necessarily enough. For example, a few requirements written loosely in a document provides less understanding than a few requirements written in a document that is backed with a model or 3-D view of the system. Such traceability of information has much more useful meaning to the designers of the system with visual representations.

Negotiating technical information and correct requirements are difficult across a team as strong communication is crucial to making progress. In a global team, the hindrance to progress is even higher. In [27], a case study for an enterprise software company is examined, where the team is globally distributed and must negotiate requirements with each other. Tools were used to communicate (videoconferences, emails, telecons, and netmeetings) as well as tools such as Powerpoint, Excel, and Word to capture information and share with other members of the team. There were a number of challenges in doing this approach: inadequate communication, knowledge transfers between groups, cultural differences across sites, and inconsistencies in capturing information. These problems emerged inadequate technical data and requirements.

Resolving the correctness of specifications requires correct technical content and data to do so. The ability to develop and capture correct requirements is an important aspect to Requirements Engineering.

## 4 Problem Analysis

The difficulties involved in effectively creating requirements are an ongoing problem. Though numerous approaches have been recommended and evaluated in the research literature, there are considerable root issues that persist making the development of requirements a significant problem in system development. These issues manifest themselves in a variety of ways and are called out in the evaluation, specifically when they appear in an application of a requirements engineering technique (see Section 6).

### 4.1 Dispersal of Critical Information

The dispersal of critical information throughout a project brings a number of problems. These problems are that information comes from a number of different sources, information is captured inconsistently throughout the project, and the impact of changes made in one area of the project may not be realized in other areas of the project.

Pertinent requirement information can come from a number of different sources and places. Information can either come from the experts on the project, or it can come from a previous design of an already built system. Previous knowledge could force the project to conform to existing information that may not be applicable or correct. Information from a previous project might not resolve with the current information of the project.

Another complication with dispersed information is that different expert disciplines in the project capture different pieces of information. The complication is that the information may or may not be relevant to the design. This leads to an inconsistency in the information being captured because different stakeholders with different backgrounds will capture information in their own unique manner[5]. While collaborating with other system experts is desirable, each expert offers different information and resolving this information is difficult. An architect and a programmer, for instance, speak different “languages” - one thinks in terms of the “big picture” and the other thinks in terms of code structure and implementation.

Furthermore, these different experts in the project capture information at different levels of detail. For instance, code can be embedded with significant details that is relevant to the software’s architecture. This information is crucial and scattered, where resolving the differences between code and an architecture may not be simple.

The impact of changing information in one area of the system might not be realized immediately, which compromises the integrity of the system. While on the surface this may not seem to be insurmountable, the impacts scale to the size of the system. A change in code could trigger a change in a requirement. Since large-scale systems are complex, these changes can have a larger impact to the system concept. This instability of critical information could correspond to missing requirements or an incomplete implementation.

The scattering of crucial information in the project may create an inability to assess the completeness of requirements[68]. The distribution of this information, and searching for the associations and relevance of this information, causes a lack of integrity in the information that can be captured as requirements. Moreover, because this information is gathered by different sources in different manners, the information can be come inconsistent and out of sync. Therefore the completeness of the requirements is compromised.

### 4.2 Articulation of Concepts

The concepts of a system are supported by the design decisions made about the system. Design, by definition, is not an isolated phase in software and system development[86] because many aspects of software development are considered as part of design decisions. These aspects include stakeholder views and concerns for the system, architectural style used for the system, post-deployment and

maintenance issues of the system, and code structure. Because many different issues and choices are involved in design, articulation of these concepts can be difficult to capture.

The concepts of the system, the architecture, and the design choices made about a system are collected in a series of artifacts that are created and maintained by different experts involved in the system. The information maintained may be difficult to trace from one area of the project to another because different information is captured by each expert and may not be formally maintained or kept up to the date. Therefore, there lies a difficulty in providing clear traceability between such strongly associated pieces of information[38, 6, 79].

For instance, an architect keeps track of a number of architectural concerns such as styles, components and connector information, interfaces, and configurations. A programmer will implement this information though there is great concern of architectural drift once implementation has begun. Code structure also carries its own design and can differ from the architecture. Moreover, trying to capture the changes and evolution made between code structure and architecture can also be difficult for capturing requirements. A change in code structure can propagate throughout the system and change the requirements.

These potential information mismatches make the articulation of system concepts difficult to capture in requirements. Most of this information is informal and difficult to express. Requirements can express this difficulty where documentation may lack a significant amount of conceptual detail or poorly express the concepts of the system's design. The inability to map requirements to design can misguide decisions about the system, particularly regarding choosing a design path. Furthermore, writing these specifications is difficult[51] because of the inability to really describe what is being built.

### 4.3 Analyzability of Requirements

From a pure requirements approach of structured statements, it is difficult to correctly analyze written statements because of ambiguity in language and notations. Structured statements can lead to misinterpretation of the words themselves. Structured statements also provides little or no insight to the intent of the requirement and the consequences of those requirements.

It is difficult to assess the feasibility of requirements in a specification because of difficulty in interpreting the language and wording of the words themselves. There are a variety of specification languages and notations that range from formal, semi-formal and informal languages. There are a number of examples, such as logic notation, natural language, and Z-notation. These different languages have different expressiveness and reasoning. This leads to misguided interpretations of the statements, and the inability to assess the intent of those statements. One attempt to solve the problem of writing understandable requirements is to follow standards for writing structured requirements documents (examples include [1, 46]). These standards however do not provide a true requirements structure or model, it provides context to a particular problem[59, 73]. These standards provide an organization to writing requirements but the ability to assess and analyze a written requirement statements because the interpretation can be misguided if the statements are not clear.

Written statements also gives very little insight to the consequences of the requirements. Requirements tend to state the nominal expectations of a system property without clear insight to the off-nominal situations that can arise from executing that requirement. One example in a spacecraft example might be the requirement of the spacecraft to eject its payload once it reaches a certain atmospheric level so that the payload can continue its mission in space. There are grave consequences of doing so and many faults that can happen and therefore more assessment would be needed to understand what other requirements are needed to meet this one requirement (or what can happen if it fails). From reading a standalone requirement "The spacecraft shall eject its payload" does not give adequate information to predict the consequences of doing so. The requirement engineer has to ask not only how such a requirement could happen, but also has to ask questions like: What are

the environmental impacts of ejecting this payload? What is the timeline to ejecting the payload? What happens if it fails? What configuration does the spacecraft have to be in for it to be able to eject its payload? One standalone requirement is not enough; its clear more analysis needs to be done to answer such questions. Its also clear that a correlating design would make this requirement more meaningful.

In another example, Heninger walks the reader through writing software requirements for the Navy's A-7 aircraft[44]. This technique speaks specifically to the outline of the document, the semantics of the requirement statements, the use of some precision notation, and organization of content. Despite the precise suggestions Heninger offers, even with the suggestions of formal language and notation, it is still not entirely possible to analyze requirements.

Structured statements still leaves its users with the inability to analyze the requirements adequately to understand the impacts of executing these requirements. Structured statements give little insight to intent of the requirements and leaves us with the inability to predict consequences of requirements. With mulitple interpretations available with structured statements, its difficut to analyze requirements.

## 5 Evaluation Framework

This section describes the Requirements Engineering Techniques selected for evaluation (Section 5.1) and the evaluation criteria for the framework (Section 5.2).

### 5.1 Requirement Engineering Techniques

A number of Requirements Engineering Techniques, herein referred to as RETs, were evaluated for their support of design and system development. Nuseibeh and Easterbrook’s seminal roadmap paper provides guidance that helps categorize potential RETs to evaluate. Nuseibeh and Easterbrook provide an overview of Requirements Engineering and discuss the areas that the field concerns itself with[73]. Overall, these concerns fall into three categories: human-centered and social process, linguistics, and system conception and design.

This research is focused around the inclusion of design with requirements development. Therefore, the RETs selected are bound to the system conception and design category. The system conception and design category concerns itself with[73]:

- Interpreting and understanding stakeholder terminology, resolving stakeholder views
- Goals and objectives of the system
- Model-driven techniques
- Validating requirements
- Correctness and precision
- Implementation and adaptability

The selected RETs are techniques that fall into the above categories and were considered appropriate for this survey. Other approaches of Requirements Engineering that were not applicable for this survey fall into the remaining categories of social process and linguistics. These RETs include ethnography, conversation analysis, requirement patterns, elicitation, and cognitive techniques. These RETs techniques were excluded in the evaluation.

The RETs are described in this section. The evaluation of the RETs to the framework are described in Section 6 and summarized in Appendix A.

#### 5.1.1 Problem Frames

Problem Frames are the structuring of real problems as a collection of interacting subproblems. Each subproblem is a smaller and simpler problem than the original. Each subproblem contains clear and understandable interactions [50, 52]. In later work, Jackson describes the problem frame view as the design task to construct an artifact. In software, the artifact would be the machine that is constructed by building software that is then executed on that machine. The machine specializes in meeting a purpose, this purpose is called the “requirement[49]”. In the Problem Frame point of view, the principal parts of a software development problem is the machine, the problem world, and the requirement, as shown in Figure 5. The problem frame approach can be leveraged in a number of ways to solving complex systems.

Problem Frames was selected as a RET for this survey because Problem Frames address specific design decisions such as separation of concerns and problem definition.



Figure 5: The Problem Frame [49]

### 5.1.2 Viewpoints

Viewpoint captures a partitioning of concerns and concepts in system development. The incorporation of viewpoints are useful in separating and resolving a set of concerns that involve significant integration with other views of the system.

Viewpoints primarily focuses on identifying conflicts between different stakeholder views and attempts to find paths to resolving these conflicts. Viewpoints provides system context for requirements development in that it divides the system into loosely coupled pieces [67].

Viewpoints was selected as a RET for this survey because Viewpoints address multiple stakeholder concerns which affects design decisions of the system.

### 5.1.3 Use Cases

Use cases in software and systems engineering are a description of a systems properties and behaviors as it reacts to requests external to the system. These external requests are labeled as “actors” and come in forms such as humans, other components in the system, or notifications. The use case technique is used to capture a system’s functional requirements by detailing scenarios that the system encounters and formalizing the behaviors and responses of the system as requirements. Use cases have been extended over time to capture different pieces of information: some containing a few brief sentences, paragraphs, and diagrams ([23] outlines full details on how to write use cases at different levels of details).

Use cases are often applied to requirements development for industry projects because of the crucial information that is required to be captured. Use cases became the basis for the Unified Modeling Language[16] (UML) version 1.x, SysML, and the Rational Unified Process (RUP), all which have become industry standards for requirements modeling and analysis with use cases.

Use cases were selected as a RET for this survey because use cases is perhaps one of the most widely used RET in practice and expresses design aspects of the system through modeling and its modeling language.

### 5.1.4 Formal Language

Formal methods are mathematically based techniques for describing system properties [98] and can be a basis for testing. A formal specification language provides the notation (syntax), the domain (semantics), and objects that will satisfy the system’s properties.

Formal Language was selected as a RET for this survey because formal language addresses important aspects of overall system development such as preciseness of requirements and verification of requirements. Additionally, formal language is widely popular as an RET in practice.

### 5.1.5 Test-Driven Development

Test Driven Development (TDD) is a software development method where test cases are written before complete development code. A code base usually exists with the TDD approach, and is continuously tested with early test case development.

While this approach focuses mostly on the design and implementation of the code from a test perspective, TDD also intertwines the development of code with requirements. The assumption with TDD is that a set of requirements are already understood before development begins. During these incremental test case development and code execution, more functionality is added to the code enhancing the code’s design. Added functionality generally corresponds to added requirements to the system.

TDD was chosen as a RET for this survey because of its popularity in the requirements engineering field, and because of the techniques used with TDD such as early test case development and iterative testing of new functionality of the system for potential requirements. Early testing of the system gives insight to the testability of requirements, an important aspect of system conception.

### 5.1.6 Domain Modeling

In [50], Jackson describes Domain Modeling as the specification of a domain and modeling how this domain can interact with other domains. For example, the domain of *spacecrafts* would interact with the domain of *space environments*. The domain modeling of *spacecrafts* would include modeling the spacecraft’s capability to follow an orbital flight path (a common characteristic of spacecrafts) over the South Atlantic Anomaly (SAA), a region in the Earth’s planetary surface where the radiation is greater than any other area on Earth. The SAA, would be a characteristic of *space environments* that the spacecraft must interact with, and therefore domain modeling aids in making the interactions between interdependent domains clear.

Domain modeling is closely tied with use cases. Much of the scenarios that are described and modeled in use cases tie closely with the domain the system belongs to. The utilization for use cases in software engineering development is also extended to systems engineering development with system use cases, using the System Modeling Language (SysML), which is an extended UML for systems engineering development[95].

Domain modeling was chosen as a RET for this survey because of its ability to scope the requirements of the system to a particular area of interest (the domain). Domain knowledge and characteristics are a significant aspect to requirements and design development.

### 5.1.7 Business Modeling

Requirements are many times driven by business needs[15]. Business modeling is closely tied to requirements analysis because of the effects that business events can have on the design of a system. Business goals have a significant impact on the requirements of a system, largely in areas of user experience, product-line development, and approaches to modeling the system.

Business modeling was chosen as a RET for this survey because of the greater influences business decisions can have on design decisions for a system.

### 5.1.8 Goal-oriented Approaches

Goal-oriented requirement approaches capture the objectives of the system at different levels of detail. Goal-oriented approaches identifies goals of the system that aids in the “eliciting, elaborating, structuring, specifying, analyzing, negotiating, documenting, and modifying” of requirements and solution structure [60]. Goal-oriented approaches are closely associated with use cases, UML, and business modeling. Goals are often modeled through a number of tools, formal languages, programming languages, and specifications.

Goal models were chosen as a RET because of the ability it has to address design choices, such as separation of concerns and solution structure through goal modeling techniques.

### 5.1.9 Agile Development

Agile approaches are widely accepted as “rapid” and “adaptable” approaches to software development, particularly because of the iterative characteristics of its development activities.

The Principles behind the Agile Manifesto states that Agile approaches “Welcome changing requirements, even late in development[11].” Agile approaches do not have a stated formal requirements or design process. Instead, Agile focuses on iterative development, feature estimation and priority development, and continuous feedback from the customer in order to understand expected behaviors of the system. Requirements in Agile development are not formally stated or specified in a specification; rather, requirements are manifested through actual implementation and customer-written tests. The iterative development between the software developer and the customer is how requirements are, essentially, “discovered” though no requirements document is mandatory. The removal of the process overhead is what makes Agile “rapid” and “adaptable” through iterations, while the requirements and architecture are simply derived from continuous testing and feedback of the software.

Agile was chosen as a RET for this survey because the Agile approach involves an intense amount of communication with the customer to understand what the customer wants in the system. Agile also focuses on an intense amount of design tasks, including incremental implementations of new functions, continuous working code for frequent releases, continuous interaction with the customer community to understand expectations and desired functionality for the system. Though there is no formal requirements process in Agile development, the activities that occur in an Agile environment are mostly centered around the evolution of the design of the code structure.

## 5.2 Evaluation Criteria

The evaluation criteria are stated below as eight properties. These properties were selected for two reasons. First, they have emphasis on design decisions[86] that are crucial to system development. Examples of these properties are *mapping to architecture* and *expressibility of critical design choices*. Second, they emphasize the development of critical information needed to make good design decisions. Examples of this include *access to relevant information* and *level of detail*.

All the properties of the framework are based on the work found in the IEEE Standard for Recommended Practice of Writing Software Requirements[46], and in the work found in [73]. The IEEE standards provides guidance on applicable information to be captured for understandable requirements. [73] discusses important aspects of software and system development that influence decisions made in the requirements of a system. These properties are not orthogonal. Their associations are called out in their definitions.

Each RET is evaluated per the evaluation framework’s criteria described here. Each property is marked with a P-number. The evaluation of each RET against the evaluation criteria can be found in Section 6 and summarized in Appendix A.

*P1: Maps to Architecture:* Assesses the degree to which the RET enables mapping of the requirements specification to specific critical architectural concepts. For example, use cases maps to architecture because use cases implement a separation of concerns. On the contrary, Domain Modeling, while it can contribute to some of the solution space that directly maps to an architecture, does not directly deal with the critical information needed to address architecture such as styles, interfaces, and components.

*P2: Expressibility of Critical Design Choices:* Assesses the degree to which the RET supports expression in the specification of critical design choices. For example, use cases can express the specific sequence of actions a user must follow to accomplish some task. Business Modeling, however, does not support the expressibility of critical design choices because it cannot accommodate the language of design such as styles and structural choices of components and connectors. Per

[86], each RET is evaluated for their respective approaches to addressing design issues, including functionality, interdependencies, non functional properties, and implementation.

*P3: Addresses Testability:* Assesses the degree to which the RET supports testability of the system; that the specification produced supports the comparison of the built product with the specification. One of the critical characteristics of written requirements is that they should be verifiable[46]. Test-driven approaches, by default, address testability concerns while problem frames do not.

*P4: Addresses Validation:* Assesses the degree to which the RET fosters effective system validation[2, 35]. Validation is the evaluation of the (whole) built product against the specification. The validation of the system encompasses more than simply testing individual requirements. Integration of system and software components is required to validate the system as a *whole*; it is not merely the sum of separate testable parts. Use cases address validation of the system as the complete set of use cases together are essentially, the system. Problem Frames, on the contrary, does not address the validation of a system in its entirety.

*P5: Accessibility to Relevant Information:* Assesses the degree to which the RET fosters traceability of requirement information to all types of relevant information in a project. Much of the premise of this property is that relevant information is dispersed through a project, where relevant information is not accessible to trace to an architecture. The distributed information of these complex systems more often leads to poor communication between relevant parties[38] therefore causing many deficiencies in the implementation of the design. Use cases, for example, can maintain mappability to relevant information with other closely associated use cases. Formal languages, however, does not have accessibility to relevant information because formal language and specifications tend to be structured in semantics and notations as output, without a true reflection of other critical aspects of the system.

*P6: Impact of Changes:* Closely associated with P5. Assesses the impact of making a crucial change in relevant information in one area of the project and determining the effect of that change on the requirements of the system. Different artifacts are developed by different experts in a project. These artifacts capture different information and are maintained formally or informally. This results in overlooked impacts to other aspects of the system when there is a significant change in one area. A change in code structure, for instance, could trigger a change in functionality and requirements. The impact of this change could be grave, particularly if the change is not executed properly. Furthermore, if these artifacts are maintained with different tools that capture different pieces of information, an inconsistency in the information is also possible, causing an incorrect decision to be made about the system's design.

*P7: Producing Artifacts:* Assesses what artifacts can be produced as a result of using the RET. An artifact is a product that captures information about the system. As requirements evolve, the traceability of these requirements to other pieces of data in other artifacts also become crucial, as discussed in P5.

*P8: Level of Detail:* Assesses the granularity of detail that is captured using the RET. Each RET captures multiple levels of details, and the granularity of these details contributes to the inconsistency of information across a system. For instance, with use cases, a number of details are captured pictorially through the use of connectors, ports, and actors. This is useful for visual representations of the system, particularly in understanding the steps the user will take to cycle through the system. Agile approaches, on the contrary, capture short user stories statements to understand how the user will use the system. This leaves the developer to interpret the level of detail that is needed to be captured to implement the user story.

In addition to the eight properties above, each RET is summarized in Appendix B. Each RET is also examined for the manner in which they visually represent their information such as graphics, models, or annotations. These summaries are included for informational purposes and not considered as part of the formal evaluation criteria.

## 6 Evaluation

Each RET for this survey is evaluated per the evaluation framework described in Section 5. See Appendix A for a matrixed summary of this evaluation.

### 6.1 Problem Frames

Problem Frames are a widely accepted approach to problem decomposition and provides guidance for further establishing requirements, system expectations, and internal system boundaries. There is a symbiotic relationship between these elements of a software system and the software architecture. Michael Jackson recognized the complexity of problems[49], where the solution is equally complex. His solution to these complexities is the notion of Problem Frames, where the large complex problem could be decomposed to sub-problems and hence better managed. These sub-problems have their own unique solution and can together contribute to the overall solution of the large problem. Essentially, problem frames are a form of software problem decomposition; a similar approach can be found in systems engineering approaches.

Problem Frames look at a problem as the concept of a machine that executes software. The machine and the software share a common phenomena (the requirements) in that problem space. From a design perspective, understanding the problem space drives certain design decisions and requirements.

The impact of Problem Frames is significant because it provided foundation for the idea of software decomposition and attacking sub-problems individually instead of attacking the larger problem. Problem Frames allows an overwhelming complex engineering problem to be handled in a methodical manner and fosters manageable solutions in the larger problem's sub-problems.

The evaluation of Problem Frames for this survey is as follows:

*P1: Maps to Architecture:* Problem Frames forces a separation of concerns between the machine and the software that executes on that machine. This separation provides guidance to the concept of the system, what problem space the system is addressing, and what properties are needed from the system. In [42], an extension of Problem Frames is applied through extended uses of architectural structures, services, and artifacts. These architectural elements are considered as part of the problem domain in [42]. This extended approach of Problem Frames engages architecture in a more complimentary manner, as software architectures are part of the solution domain. From this coupled relationship, [42] looks to engage architecture as part of the requirements space.

Problem Frames is a common RET in part due to the separation of concerns between sub-problems. However, Problem Frames are far removed from details. In Jackson's earlier works[49], the Problem Frame was too abstract, and did not give specific problem decomposition guidance[80]. To address this, [80] introduces Architectural Frames. Architectural Frames takes the architectural style chosen (a pipe-filter style for instance) and translates that style to the Problem Frame schematics. The benefit of introducing Architectural Frames is that the solution space is now driven by the architecture style chosen, and less by the shared phenomena of the system (which was the original work in Problem Frames). With Architectural Frames, the sub-problems that are decomposed are driven by the architecture. These sub-problems in Architectural Frames looks at characteristics and properties of the architecture in order to derive the solution space.

*P2: Expressibility of Critical Design Choices:* Problem Frames do foster critical design choices in that, once sub-problems are clearly defined, decisions about the solutions to these sub-problems can be explored. However, in the original Problem Frame concept[49], critical design choices were not captured in a Problem Frame (architectural style, component and connector definitions, as examples). Later work defined architectural styles[80] relative to Problem Frames which offered some manifestation of design choices.

Multiple solutions to complex problems make design decisions hard to establish. In [61], the Composition Frame, an extension of the Problem Frame concept, brought forth different design solutions in the frame itself. The Composition Frame forces the problem space and the solution space to not be distinctly separated and shows various design options for a solution. Because multiple solutions can exist to a problem, the Composition Frame is described in a solution-centric construction. The Composition Frames is a requirements construct that models relevant aspects of composition and deals with unwanted side effects (for example, an overlap of triggers to system events). Composition Frames forces some of the solution space to be surfaced and does not separate the solution to the problem. Composition Frames allows for multiple solutions to be maintained.

*P3: Addresses Testability:* Problem Frames do not directly address the testability of its construction. This is generally due to the project phases that these two activities occur in: Problem Frames development occurs significantly earlier than the testing of requirements. Testing requires more details for test case development, and Problem Frames do not provide these details directly.

*P4: Addresses Validation:* Problem Frames do not address the validation of a software system. These two activities occur at the beginning and tail end of the project development and logistically, these two activities are not tight coupled. Problem Frames development occurs fairly early in development, where later phases of software development are not considered.

*P5: Accessibility to Relevant Information:* Problem Frames do support accessibility to relevant architectural information in the project, specifically with Architectural Frames[80]. Architectural Frames looks at the chosen architectural styles and translate the style into Problem Frame schematics.

*P6: Impact of Changes:* The impact of changes made in a Problem Frame does have an overall effect on the system, particularly in architectural elements or domain characteristics of the Problem Frame. A changes made in a Problem Frame is critical because overall problem to solve will be changed. The impact of change in Problem Frames can be more cumbersome if there have been additional artifacts that have been developed in the project, where that information is tightly coupled to the Problem Frame schematics. The largest impact is on requirements, particular on requirements that are based on shared characteristics of the problem's domain. One study [58] looks at the problem of domain characteristics changing and its impact on requirements. [58] suggests the use of a domain ontology to reconcile these changes where common systems share characteristics and these common characteristics can be called upon as requirements.

*P7: Producing Artifacts:* Problem Frames are captured in documentation as textual explanations, architectural descriptions, state diagrams and behavior, and contextual diagrams. Problem Frames can be further decomposed to formal specifications with a specific problem frame syntax (Composition Frames[61] uses specific syntax for specifications, as an example). Much of the Problem Frame work has informal semantics involved, including state symbols.

*P8: Level of Detail:* In [48], the informality of Problem Frames are discussed. The domain characteristics and the machine characteristics are separate and not accessible, but the machine and its domain share a common set of characteristics. These shared characteristics are usually captured as requirements and specified in formal documentation. However, the designation of common characteristics can lead to false errors in the details as some of the shared characteristics may be insignificant. Though Problem Frames capture details of the system's concept and problem, there are no guidelines to determine the relevance of these details for the system.

## 6.2 Viewpoints

The issue of conflicting stakeholder views is a concern in developing requirements[71]. The development of large complex systems involves a number of disciplines and experts that contribute to the system’s concept and development. Naturally, there will be conflicts in each stakeholder’s view on the system, particularly in areas of functionality and behavior. This poses a problem to the requirements aspect of the system, in that deriving requirements from conflicting views is very difficult.

The notion of a “Viewpoint” addresses a critical problem in the Requirements Engineering field with regards to these conflicts. In the original Problem Frame concept, conflicting views were not considered. Viewpoints do draw on the Problem Frame concept, and further encapsulates key principles of software engineering: separation of concerns per stakeholders, heterogeneity of system representations and views, and decentralization of conflicting concepts of the system. The Viewpoint approach has widely been extended to a number of frameworks, tools, and methods to dealing with conflicting views.

The evaluation of Viewpoints for this survey is as follows:

*P1: Maps to Architecture:* Viewpoints utilize the concept of Problem Frames in that the viewpoints further separate out stakeholder conflicting views so that they can be addressed. The architectural benefits of doing this is known as “delayed commitment[31]” in that alternate representations of the system are explored and modeled around a specific viewpoint before any architectural choices are made.

The Viewpoint approach has been used inconsistently. Most studies and theories have discussed Viewpoint usage in the elicitation phase of requirements work. In [31], two teams were divided to look at an industry project and explore the requirements modeling needed to design the system. One group used viewpoints for model planning and the other team merged all models using i\*. It was found that the team using Viewpoints were able to resolve some large issues than the team that used i\* modeling, most notable with the mapping to the architecture model sizes. The difference between the two teams were significant, largely due to the fact that the Viewpoint team was able to break the different viewpoints into manageable pieces.

The Viewpoints approach focuses on representation of consistent information while tolerating the differences between these views. Because requirement information and data passed through the project are generally inconsistent between stakeholders, the multiple viewpoint approach allows for this flexibility for inconsistencies. Such flexibility addresses design concerns and become design drivers in system structure.

*P2: Expressibility of Critical Design Choices:* Conflicting and different viewpoints are design drivers to the system, especially in limitations that can exist in the construction of the system. In [86], “views” and “viewpoints” are centered around the set of design decisions based on a set of concerns. Each of those concerns ultimately drives the viewpoint. [86] looks at Viewpoints from an architectural style perspective, in looking at a particular architectural style as a filter for the relevant information of a key view of the system.

Resolving these conflicting views fosters a decentralization of concepts[71]. Decentralization of concepts enforces a standard set of properties for the system to adhere to. In cases where large system development is utilizing completely new technology (as opposed to reuse of old concepts, systems, or standard technologies), the decentralization of these concepts may diminish the quality of good design. This forces the development to reuse the same recycled information from a previous system without innovation. While for some systems, this might be the choice in the interest of time and limited resources, this may not be the choice for new technology development.

[31] looks at Viewpoints as a matter of performance, where scalability became a key concern in the development of a particular counseling phone service data site. The case study was of a

complex project that involved some major inconsistencies with stakeholders. One of the major takeaways from this case study was the importance of maintaining a tightly coupled relationship with the product and its design. The case study was large and complex, and the scalability of the conceptual models were crucial in achieving successful understanding of the system.

In another study[67], a data site was explored where requirement modeling were derived from different viewpoint models. The authors describe tolerant inconsistencies that can be overlooked in the system's design. This tolerance is interesting in that it allows some flexibility in the design of the system and tolerates flaws in the system and its design. It was shown in this study that each respective inconsistency had similar connotations but are expressed differently in terminology and in requirements. This is an example of a tolerant inconsistency impacting design decisions of a system.

*P3: Addresses Testability:* Viewpoints tend to not address testability issues in its approach. Viewpoints explore design decisions and alternate design choices of a system and does not focus on the testability aspects of these design choices.

*P4: Addresses Validation:* Viewpoints tend to not address system validation issues in its approach. Viewpoints explore design decisions and alternate design choices of a system and does not focus on the testability aspects of these design choices.

*P5: Accessibility to Relevant Information:* Viewpoint information captures relevant details about the stakeholder's view of the system. If all the stakeholder's system views exist in a common environment and the information captured for these views follow a standard, then the information contained in these views is easier to access and allows the focus to be on the information contained in the Viewpoint.

Easterbrook's tool *Synoptic*[29] is intended to aid the requirements work by emerging conflicts between the different views of the system. Participants who use *Synoptic* externalize their conceptual model in a common environment. By making these models available to other participants, the conflicts are realized sooner and examined for inconsistencies. Mismatches between stakeholder's models are categorized and options for resolving these mismatches are generated in order to proceed forward. In the *Synoptic* tool, relevant information comes with the inconsistencies in stakeholder views. With Viewpoints, collaboration of these views (particularly in a common environment) are necessary in order to resolve conflicts.

*P6: Impact of Changes:* For Viewpoints, change tends to occur when information in a view changes. The impact of these changes can have a positive impact if the change causes a reconciliation between conflicts. If these views can exist in one common environment with easy accessibility, the impact of change in the Viewpoint information can be realized quicker.

*P7: Producing Artifacts:* In most projects, Viewpoints are captured in a number of formal artifacts, either design specifications or architectural descriptions. Viewpoints are also modeled[31], largely to resolve the inconsistencies between two or more models that represent conflicting approaches to the system.

*P8: Level of Detail:* Viewpoints capture details that are tailored to particular views of the system. There are no standards that exist on what details are captured in viewpoints.

[84] argue that previous work done in Viewpoints(such as [71, 74]) shows that viewpoints are too structured and rigid. [84] suggests a more flexible ways of developing viewpoints. This flexibility, instead, is driven by the user in defining their own viewpoints. In this case study, a number of experiments with industry applications were performed where the participants had the flexibility to create and develop their own viewpoints of their system. The general observations were that viewpoints helped start the elicitation process but eventually the developers derived their own

approaches to better understand the system. However, there are some severe limitations in this case study. None of the studies were interested in versioning or history changes in the requirements to maintain a record of contents and its changes. Furthermore, the case studies in this work did not end up using all the viewpoints that were derived.

The usefulness of details captured in Viewpoints is dependent on the content that the viewpoint is capturing and filtering out[86] in order to proceed with good design decisions.

### 6.3 Use Cases

Use Cases is a visual modeling technique for software development. With use cases, structured scenarios are developed using natural language, where the interactions between different scenarios, users, and actors are described.

Use cases are unique from a requirements perspective because the information that use cases capture provide requirements. The usages of the system are modeled, the actors and internal behavior of the system are captured, and interplay between closely associated use cases provides great context to the requirements of the system.

The evaluation of Use Cases for this survey is as follows:

*P1: Maps to Architecture:* Use cases have a strong mapping to architecture when proper decomposition has been met. Use cases show a strong separation of concerns between different use cases, actors, and interfaces. The key properties required for input with use cases can separate the functional behavior properties of the system. Understanding the syntax and semantics of UML is important to actually make use cases useful in architecture descriptions. Use cases offer a preciseness needed for architectural components and decisions.

This preciseness however is often proportional to the implementation of the use cases in order to determine its ability to map to an architecture[86]. The information needed to trace an architecture to a use case can be incorrectly implemented. In [20], Use Case Maps (UCMs) are presented as “architectural entities” in order to gain insight to all developmental phases of the system. UCMs involve a “lightweight notation” but claim that the insight to the system’s behavior is “in the eye and mind of the beholder” (and mostly inferred through heuristics). The UCMs depend on the imagination of the user to visualize data or information being moved in the system by actively moving tokens. There are severe limitations to this study, namely in that the important behaviors of the system (data moving in the system, system behavior paths) are to be inferred by the physical location of the data in the UCM notation. The syntactic rules are user-driven, as is the interpretation. Such dependency on human inference is an example of misinterpretation to mapping use cases to the system architecture; in fact, this case study allows too much ambiguity.

Use cases can come with many problems. [64] discusses ten most common pitfalls in deriving use cases. These pitfalls include misunderstanding of system boundaries that lead to incorrect use case definition. This misunderstanding is often tied to an incomplete architecture or misinterpretation of the design decisions of the system. Not understanding the system scope leads to a misdiagnosed use case (Pitfall # 7) and an incorrect functional description of that use case (Pitfall # 8). Fowler further calls the mis-use of use cases[36] namely around incorrect decomposition.

*P2: Expressibility of Critical Design Choices:* When implemented correctly, use cases supports the expressibility of critical design choices for the system. Use cases adhere to defined structures for critical design elements.

Use cases also have a number of setbacks when implemented incorrectly[86, 64, 36]. [86] suggests to not overestimate the benefits of UML as a means of capturing design decisions and using use cases as a design approach. This is largely due to the ambiguity in the semantics of UML, as well as the fact that there are no formal standards to evaluate the correctness of use cases. While the

information captured in a use case has a multitude of basic elements associated with design drivers (classes, associations, states, activities, events, constraints, and scenarios), there are many problems in the correctness of the information and its association with design decisions.

*P3: Addresses Testability:* Use cases shape the test approaches of the system, by drawing on the use cases themselves as the basis of test cases. One of the major benefits of use cases is that the details captured in a use case offers a testability aspect of the system, because the usages of the system are captured in a use case. Knowing the usages of the system provides a basis of what needs to be tested[69]. Where as simple “shall” statements can be ambiguous, use cases avoids some of this misinterpretation because the usages of the system are better captured.

Linking a use case and a test case is not simple. [70] aims to resolve the gap between design use cases and test use cases by proposing an approach for automating the generation of system test scenarios in an embedded software project (in this case, an aircraft, a typical large-scale system). The proposed approach takes into account traceability problems between high-level views and concrete test case execution. A transition system was built between a formal requirements specification to a test case output.

*P4: Addresses Validation:* While there is not a direct link between the overall validation of a software system and use cases, there is an association between these two software activities. Use cases manifest the interplay of actors and usages of the system. The integration of these use cases function as insight to the validation of the system.

*P5: Accessibility to Relevant Information:* Crucial design and architectural information is captured in a use case. The accessibility to this relevant information is easy and possible to do because use cases exist in a common environment and associations between these use cases are easy to access in the same environment. Therefore, when one piece of information changes, the rest of the use case or use cases are updated as well. Dependent on the tool used, much of these changes can be realized sooner.

*P6: Impact of Changes:* The change in a software’s use case can be realized sooner in the development process than later. Use cases have a direct linkage to other associated use cases. While the impact of change is costly, the realization of changes quickly emerge in development.

*P7: Producible Artifacts:* Producible artifacts for use cases are the use cases themselves, scenarios and activity diagrams, sequence flow charts, state behavior, and requirements specifications.

Use cases, in many examples, are copied into an actual requirements specification and mapped directly to a requirement statement. Lilly discusses this as a potential pitfall of using use cases in specifications[64] (Pitfall # 7), as the confusion of including them makes it difficult to understand the actual requirement. This constraint can be addressed in some cases with the specific tool of use. For example, with the popular requirements repository tool, Telelogic’s DOORS[87] does come packaged with Telelogic’s UML tool Tau[88] where use cases directly trace to the specific requirement in DOORS. A formal traceability matrix artifact can be produced from this linkage.

*P8: Level of Detail:* A significant amount of design detail is captured in use cases. In [23], Cockburn describes three levels of use cases, each with different degrees of detail: informal (consisting of a few sentences summarizing the use case), casual (consists of a few paragraphs of text, summarizing the use case), and fully dressed (formal specifications based on significant detail based on the key properties discussed earlier in this section).

In [89], an evaluation criteria for assessing the quality of use cases is proposed for the Volvo automotive system. The evaluation criteria proposed in this study was a set of questions rather than utilizing the standard use case template. These questions assessed correctness, consistency,

ambiguity, completeness, readability, and level of detail. The results of this study were skewed based on human input and missing elements such as: common terminology taxonomy, utilizing use cases as black boxes and not integrate these use cases with scenarios (dealing with scope issues), and overloaded design details (specifically on how parts were designed). This level of detail deterred the focus on functionality. The most common pitfall in this study was the simply misuse of *precondition* which ended up propagating through later design decisions and details.

## 6.4 Formal Language

Formal language notation, specification, and formal methods support design rationale in that the accuracy and measurement of specific structures and interfaces can be assessed through the use of this RET.

Several misconceptions about formal languages and methods exist about the uses of using this RET for requirements development. One supposed cost is that formal methods eliminate the need for testing because of the mathematical preciseness in proving program structures correct[40]. Formal language and methods cannot replace testing activities or any other traditional engineering design methods[17]. [17] argues that the use of formal methods in design actually enhance the possibilities, especially in requirements analysis where an alternative view can be presented.

Formal language and notation requires a steep learning curve, which many users find unacceptable [46]. This is partially due to the difficulty of understanding formal semantics or a general confusion on how to apply formal methods to design.

The evaluation of Formal Language for this survey is as follows:

*P1: Maps to Architecture:* The ability of Formal Language to map to an architecture is possible through a collection of semantics and type-checking implementation of architectural components[86]. Architectural styles, however, are difficult to reflect in formal notation.

[4] discusses the increasing complexity of software systems stating that the overall system structure is the software architecture. The interactions of components and connectors in this study are defined in explicit semantic entities. The entities are captured as a collection of protocols that characterize each of the participant roles in an interaction and how these roles interact. The formal semantics maps the system's architecture to comparable programming implementations, where compatibility is resolved.

*P2: Expressibility of Critical Design Choices:* The formal specification through the use of formal languages describes critical functional properties as part of the design efforts of a system[22]. The formal specification expresses system behaviors in areas such as timing, performance, accurate internal structure, and other key functional properties.

Though there are not an abundance of real-world empirical examples of the application of formal methods as a design approach[30], one notable example was the ATC Information System[41], as part of new air traffic management for London's airspace. The requirements analysis phase of the large-scale project was developed using formal descriptions to structured requirement notations. The end result found that the overall production effort using formal language and specification was equivalent in time and overall better quality in the requirement and design work. Another example with a successful outcome through the use of the Formal Language RET as a design implementation can be found with a study done on the Lockheed C130J aircraft[34].

In another study[94], a formal notation (Object-Oriented Structured Design, OOSD) was developed to support key software structure concepts and design principles. The notation allows the representation of modules, interfaces, hidden information, concurrency, message passing, and operational behavior. The design approaches that were examined were object-oriented design, functional decomposition, and data structure design.

*P3: Addresses Testability:* In earlier work [40], a common misconception is that the use of formal methods can take place of several testing activities. Formal methods support testability of requirements as it provides a quantitative approximation of the real environment the system has to function in. Formal methods provides proof of measurable properties and performance[98].

*P4: Addresses Validation:* The formal specification offers insight into the validity of the design approach through the emphasis of proof of correctness of system properties.[22] discusses the value of formal methods for software system validation, in that theorem or mathematical validation of the system is crucial in system buy-off.

*P5: Accessibility to Relevant Information:* Does not apply to the evaluation of Formal Language. The criteria of accessibility refers to the mapping of relevant design information. Formal methods are mathematically based with theorems and proof of correctness of requirements. Any change to pertinent system information would be reviewed manually by the designer.

*P6: Impact of Changes:* This evaluation property also does not apply to the evaluation of Formal Languages.

*P7: Producing Artifacts:* Formal languages are primarily used with formal specifications of requirements by providing accuracy and measurements of correctness[22].

*P8: Level of Detail:* Formal languages in specifications provide significant detail to the properties the system must meet. There is a belief however that the formal methods community has overemphasized full formalization of a specification or design[47]. The expressiveness of the languages involved, as well as the complexity of the systems being modeled, make full formalization a difficult and expensive task. Several “lightweight” formal methods have been proposed as an alternative, which emphasize partial specification and focuses on design aspects of the system not closely tied with mathematical measurement.

In [86], architectural models are placed in three categories: the informal models (captured in box-line symbology), semi-formal models (most notably, UML), and formal models (with formal notation as a basis). There are degrees of difficulty and precision issues in all three, all dependent on what level of detail is desired.

## 6.5 Test-Driven Development (TDD)

TDD approaches is early testing in development and can influence design decisions of the system. By developing test cases early in system development, system functional properties can be determined or re-assessed.

TDD approaches have directly influenced the work of requirements because it addresses the feasibility of the requirements being considered for the system. The use of TDD directly influences the design[55] with regard to programming aspects as well, because TDD fosters early code development. Weighing functional properties and their feasibility in early system development through testing these properties can motivate good design decisions to be made about the system.

TDD development emphasizes early test cases to be written before full functional code. Some of this development can shape the code structure itself in determining what is testable. early results of a preliminary test case can result in more informative design decisions.

The evaluation of Test-Driven Approaches for this survey is as follows:

*P1: Maps to Architecture:* TDD encourages test case development before a solid code structure; therefore the architecture of the system can be lost in an unstable code base. Since TDD is an iterative approach, the code base and the test cases incrementally grow as well. As the code base and test cases grow, it is possible to drift from an established architecture. A strict adoption of the TDD approaches can result in “missing the complete picture of the software[12]”.

*P2: Expressibility in Critical Design Choices:* TDD does express certain critical design choices made for a system[55] because it places emphasis on the feasibility of certain functionalities and system behaviors. TDD supports an evolving of implementation design because writing tests prior to code enables choices to be made about software’s behavior and functionalities[56].

Placing emphasis on early testing of working code while dismissing critical design choices to be considered and captured before implementation starts[86]. Design can be lost with early testing of code that is continuously evolving.

*P3: Addresses Testability:* TDD facilitates testing activities. There is conflicting literature however, that debates whether TDD is a primary testing strategy or a primary implementation and requirement strategy ([55, 9, 86] and many others that take opposing views on this).

The benefits of TDD are that the test cases developed that are maintained regularly for feedback to testable code[65]. As code is updated and maintained, test cases are executed to evaluate new functionality built in the code which provides feedback to newer requirements that are testable. TDD forces the developers to keep up to date test cases for the code meaning that the code itself will likely be better understood if its constantly being tested. Defects in the code is discovered and resolved sooner where the test cases are available for code testing activities of the system[9].

*P4: Addresses Validation:* While TDD addresses testability aspects, there is an implied tie to system validation. Constant re-testing of newly developed code gives insight to the validation of the system.

*P5: Accessibility to Relevant Information:* TDD fosters documentation in the test cases and code base. Hailpern and Tarr assert that there is difficulty in preserving various documents written at different times of the project and that the “meta-information” associated with these artifacts might explicitly relate to nothing else in the system even though the relationships are existent and well understood[39]. The authors also contend that it is difficult to understand a test case much later in the project and understand why it was applicable to the system in the first place. There is no perscribed approach to assessing pertinent information using TDD.

*P6: Impact of Changes:* With TDD, it is most critical to ensure that the test cases and code base are current and up to date. Therefore a change in a test case can trigger changes in requirements and overall change in code structure. Hailpern and Tarr[39] argue that it is a challenge to convince developers to create and maintain information to start with because it costs time and resources that can be used elsewhere in activities, particularly with test cases.

*P7: Producing Artifacts:* Producing artifacts for TDD approaches are the actual test cases and the code base.

*P8: Level of Detail:* There is significant details embedded in test cases; however there is other critical information that can be lacking in the documentation of test cases[12], including the rationale of a test case’s existence. Test cases do provide a basis of measurement of accuracy of requirements implemented in the code, and can provide insight to measured progress (i.e. number of test cases that passed versus ones that did not).

## 6.6 Domain Modeling

The domain space of a large problem namely contains the key characteristics of that domain, and therefore the products that are built under that domain also share a common phenomena of behavior. The domain model is a representation of the world that the machine must exist in, where the overlap of the domain and machine characteristics are the requirements of the system. The relationship of domain characteristics, machine characteristics, and the characteristics they share (the requirements) are shown in Figure 6.

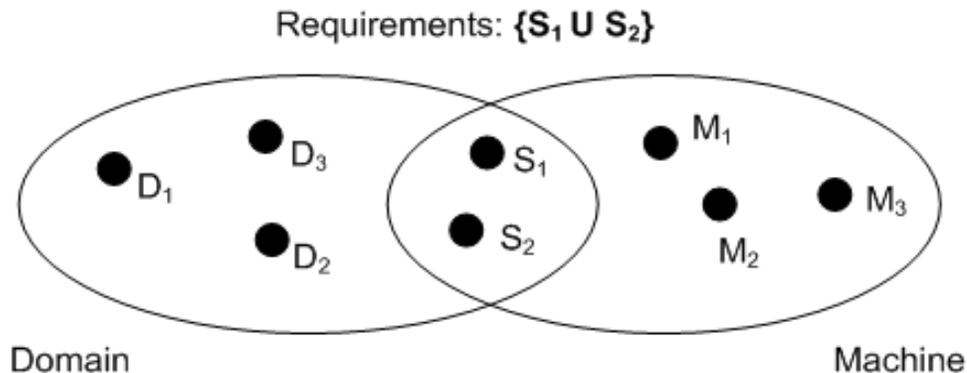


Figure 6: The Domain, The Machine, and Shared Characteristics[48]

The evaluation of Domain Modeling for this survey is as follows:

*P1: Maps to Architecture:* Domain-specific software architectures is made of a body of knowledge about a specific application domain[86]. Domain models capture key properties of the domain, specifically manifesting what happens in an application. Functions that the model executes as well as the specific entities performing these functions are captured in domain models, providing traceability to an architecture.

In [80], the authors observe complimentary views in the solution space, all centered around domains and architectures: one, the solution will be driven by the developer's domain knowledge; two, the choice of domain architectures, architecture styles, and design patterns are usually considered for domain modeling; and three, the developer's past engineering experiences influence decisions made about the system (issues with reuse of previous system's architecture can be found in [19]). Domain modeling, and specifically, calling on domain knowledge to appropriately construct the system is a common first step in decomposition; thereby the mapping of domain modeling and architecture does exist.

*P2: Expressibility in Critical Design Choices:* Domain models is expressive in capturing rationale behind critical design choices. Domain models come with a number of taxonomies and terminologies that are common for products in that domain, therefore providing a standard for the given domain's nomenclature and characteristics[86]. Domain models also provide standards of the descriptions of the problems that need to be solved in that domain. These descriptions provide insight into domains and their associations and captures traceable reasoning for particular design decisions.

*P3: Addresses Testability:* The Domain Modeling approach does not address testability specifically. Testability is not applicable to this survey's evaluation of Domain Modeling.

*P4: Addresses Validation:* The Domain Modeling approach does not address validation specifically. Validation is not applicable to this survey's evaluation of Domain Modeling.

*P5: Accessibility to Relevant Information:* Domain models provide a number of critical pieces of relevant information about a particular domain. Domain models provide dictionaries, terms, and important information that is relevant to architectures. Domain models contain characteristics of a domain that are common to the products that are developed in that domain, therefore accessibility to critical information in a domain model is not very difficult. For instance, the domain of spacecrafts has a number of common elements to it (instrumentation, onboard attitude control, solar panels, as examples). These common elements are easily accessible for potential design decisions and requirements. In fact, one study[58] suggests an ontology of domain characteristics for requirements development, where relevant information can be maintained in one place.

*P6: Impact of Changes:* Changes made in a domain model has a strong impact on the changes made in requirements. If a key characteristic of a domain model changes, and a number of architectural elements are dependent on the information in that domain model, there is a significant impact on requirements. Should a number of changes occur in a domain model where its meaning and intent is altered, the impact of these changes can be quite costly to rework of requirements.

*P7: Producible Artifacts:* Domain models are generally represented in modeling languages such as UML. Domain models also have a number of associated artifacts with it: domain-specific dictionaries, domain knowledge, and domain-specific architectures.

*P8: Level of Detail:* Domain models can vary in abstract detail to concrete detail. In [53], much of the domain models are translated to formal notation from graphical representations. Domain models are tailored to the needs of system decomposition. Domain models can also be precisely built with UML.

## 6.7 Business Modeling

Designing a large complex system is done in stages of effort. Design decisions can be bound with the implementation of the system without an understanding of the business aspects a system supports[66]. The decisions that affect the business can affect its software implementation because software supposedly is the easiest to change[19].

As an example, the business model for banking very much affects the software and banking systems that are developed for the financial industry. There are specifics to managing finances and spending money for example that are very tied to the business model of banking. Because of these business characteristics, any system designed to address the banking activities of a consumer must profoundly understand the financial business and its problem space.

There are a number of programming languages and tools that specifically look at modeling a business' problem space (and UML, for instance, is quite popular for business modeling). Business modeling allows for alternative business implementations to be considered and exposes effects of the business to the systems developed for the business problem space.

Traditional requirements specifications written in natural language often do not address the industry aspect of the system; and in fact, should not. Depending on the background of the customer, the linkage between the business aspect of the project and the system proposed to solve some problem in that aspect is crucial. Business modeling by no means a substitution for the work involved in designing a system.

However, Business Modeling is addressed in this survey because of the insight it brings to the system for the business that the system supports.

The evaluation of Business Modeling for this survey is as follows:

*P1: Maps to Architecture:* A strong architecture can greatly influence the business model. Business Models, however, do not directly map to an architecture style or architectural elements of a system.

The mapping of a Business Model and its architecture should be a correlation between a successful product and a clearly defined architecture. A lot of products these days market their success on clearly defined processes which is not the heart of a product. While clear processes are one pillar of success and an organization, the heart of the product lies on the integrity of its construction. The quality of that product should nominally be centered around its architecture[86].

*P2: Expressibility of Critical Design Choices:* Design choices of a user interface and interaction with users can be driven by the business model.

Business needs can impose design limitations on a system. In one case study[15], a study was conducted on the Syrian Tourism website. It was not clear to the researchers what the goals of the business were, so implementing the business values of the Syrian Tourism office was difficult. Clarity was needed before the researchers could implement the website system. Goals were derived into a set of requirements that were tangible and would ultimately drive the design of the website that the user was expected to experience.

Business goals and values are reflected in the design choices made of the system. Design choices requires interaction with customers, marketing, and business models to understand what consumers expect from the system. Business models can impose standards for designs of the system specifically tied to market demands and consumer expectations.

*P3: Addresses Testability:* The Business Modeling approach does not address testability specifically. Testability is not applicable to this survey's evaluation of Business Modeling.

*P4: Addresses Validation:* The Business Modeling approach does not address validation specifically. System validation is not applicable to this survey's evaluation of Business Modeling.

*P5: Accessibility to Relevant Information:* Accessibility to relevant information is not applicable to this survey's evaluation of Business Modeling.

*P6: Impact of Changes:* One major impact of changes comes when the business model and processes change. This can directly affect the implementation of the system or products of systems that support the business. This was shown in one case study[54], where the internationalization of a small product caused requirement approaches to the product to completely change. The product had to scale to a larger, international audience and required an increase in personnel to accommodate the change in the business model of the product.

*P7: Producing Artifacts:* Artifacts of business modeling usually result in business use cases and in some examples, a business or marketing requirements document.

*P8: Level of Detail:* Business modeling is considered rather abstract in detail in terms of the system's concept; the focus is around the business problem space that the system is addressing, and less about the detailed specifics of the design.

## 6.8 Goal-Oriented Approaches

A *goal* is an objective that the system should meet. Goals provide guidance to the design being considered, particularly in understanding what the system needs to do and what the scope of the system.

The development of goals is a human-driven process; language semantics and conflicting viewpoints can make the development of goals difficult.

The evaluation of Goal-Oriented Approaches for this survey is as follows:

*P1: Maps to Architecture:* Goal models maps to an architecture in that it provides rationale to a particular architectural style selected for a system. Goal models can also capture expected interfaces between a system and external factors which can also be reflected in an architecture.

[91] provides examples of architectural descriptions that are derived from goal-models. Using the component and connector description language, authors use goals to derive the behaviors of the architecture (connectors behaviors, components behaviors, for instance). [91] attempts to “architectualize” the non-functional aspects of the system (namely parts of the architecture that you would not necessarily consider an imposed requirement). The authors in this study integrates multiple models that are built in an incremental fashion, bringing consistency of goal-models to architecture models. The non-functional goals from the goal model are used to refine components and connectors. The refined architecture (written in a particular ADL) can now validated against the requirements model already presented. The development process of developing these ADLs from goals is iterative. There are a number of constraints with [91], namely that the mapping to architecture components and connectors are informal therefore there is a loss of intended capability when moving from goal models to an architecture.

*P2: Expressibility of Critical Design Choices:* Goals encapsulates a number of design choices that can include interactions with humans and users, devices, and the environment that the system must be compatible with. Goals can also include the non-functional or usage properties, where multiple design elements for instance, must interplay together to meet a system goal. As an example, an online banking system may have a goal of providing secure transactions if a user wants to move money from one account to another account. However, to meet this goal, the system will rely on multiple software items to cooperate together to ensure the security of this transaction. These goal details provide guidance to design implementations to meeting that goal.

Goals can capture a number of common terminology and usages of the system that can support design rationale behind certain design decisions. An approach is presented in [57] called the Goal Argumentation Method (GAM) which captures the decision rationale process to decide on goal models. GAM is particularly useful when interdisciplinary stakeholders are involved and decisions must get recorded for completeness and reference. Using GAM, the stakeholders construct arguments to justify modeling decisions and use these arguments to question existing arguments and decisions behind them. The arguments can lead to clarity in developing requirements or offer potential requirements that have yet to be discovered. What [57] shows is an approach to design evaluation where potential variants in design rationale is captured as part of the goal and propagated through the system’s development with output including a graphical representation and formal notation. This study emphasizes the need to keeping information less dispersed and map to design rationale of goal implementation.

*P3: Addresses Testability:* Goal modeling does not directly support testability of goals or testing activities. There are no standards to test goal models. This property does not apply to Goal Modeling.

*P4: Addresses Validation:* Goal modeling does not directly support the validation of the system. In [60], goals are reasoned to be validated by providing scope in how the goals play out, with the idea being that all these goals will collectively validate the system and meet the systems goals. However, there is no true support for this assertion. For this survey, validation is not a property addressed with goal modeling.

*P5: Accessibility to Relevant Information:* Goals models capture relevant system information and

accessibility to these goals are not very difficult.

*P6: Impact of Changes:* An change in a goal trigger a change in concept or design.

*P7: Producible Artifacts:* Producible artifacts in goal modeling includes goal models, goal use cases and specifications that are derived from goal models[60].

*P8: Level of Detail:* Goals are formulated at different levels of abstraction. Goals can be abstract and contextual in nature, or they can be strategic in the concerns they address. Goals cover functional concerns, services needed from the system, and non-functional properties (i.e. accuracy and performance).

## 6.9 Agile Development

The “Agile Movement” focuses on a few core values for development. First, the Agile community places priority on communal software development where the working team collaborates closely. Second, the relationship between the team and the customer is priority, where the customer receives valuable software often, and the customer’s feedback is incorporated in development. Agile addresses customer concerns and integrates the customer as part of the development efforts. Therefore, customer feedback is received sooner and the feedback is considered very valuable. Processes are not part of the relationship with the customer, in fact no formal processes are required with Agile. The Agile Manifesto calls on face to face meetings internally, continuous customer meetings, continuous releases of working software, close team rapport, and physical co-locality of developers. Third, the development team releases working tested software frequently. The frequency of these releases ranges from days to a few weeks (in some cases labeled as “sprints”[83]), are considered relatively short. Lastly, the more crucial value that the Agile community follows is embracing change, allowing requirements to change late in development, and adapting to changes in development or ideas.

Such values poses difficulty in large-scale complex work, particularly in defense work where short sprints are not always possible due to schedule milestones and the contractual relationship with the customer. The customer generally does not have the resources to continuously participate in the development effort, and processes are part of procured work. However, there has been many successful cases of Agile methods used in such line work where the Agile processes are tailored[26] to meet the needs of the contract.

The evaluation of Agile for this survey is as follows:

*P1: Maps to Architecture:* Overall there is no formal specification or architecture descriptions captured with Agile. The working code is considered the architecture as the Agile development process does not promote the development of an explicit architecture. Though architectural description languages and styles are not commonly called upon in Agile development, Agile does support architectural support and inclusion in its code base instead. [13] suggests that because Agile emphasizes refactoring and constant change, architecture must keep up with these changes as well. XP specifically filters out unnecessary features with feature estimation and giving priorities to implemented features. In large-scale work, some of these unnecessary features can become future requirements for future systems.

*P2: Expressibility of Critical Design Choices:* Agile development does support some expressibility of critical design choices made about the software system. The concurrence of development activities does support some inclusion of design decisions. Agile emphasizes early testing of working code which influences elicitation of requirements; these parallel activities encourage design choices of the

system. An established code base however, could be extensive in documentation but no explicit capturing is made of the design principals that the code manifests. Therefore, design decisions and rationale can be lost in the code base.

Design properties of a system also focus on stakeholder concerns. Agile focuses on stakeholder concerns with user stories[25] to better understand how the users want to use the system. User stories capture what the user expects from the system. Agile developers incorporate these user stories in the code base. Addressing usages of the system is another important aspect of design.

With Agile, design properties of a system can be found in the working code itself[75], particularly with XP where the emphasis is on the code, testing of code, pair programming with co-developers, gaining customer feedback, and design implementation as part of continuous coding. Agile values working increments of software over process and documentation. Features are prioritized per release (as what is done in Scrum sprints for instance) where the focus can be on the needs of the user, incorporating design with coding, understanding what the customer is expecting, and overall developing the system. Features that are implemented through programming allows for the design and implementation to be intertwined. What results are the understood requirements from the continuous iterations and releases of the code.

In another study[90], the authors discuss the number of assumptions and limitations of Agile development, in particularly with the Principles behind the Agile Manifesto. One of these assumptions is that visibility into progress is through working code. For systems that involve user interfaces and requires user and customer feedback, this visibility is very valuable. However, with embedded software systems (often found in large-scale projects), working code does not necessarily provide insight into the design of the system, largely because software is embedded in hardware and by the time a user interacts with the system they are far removed from the code.

*P3: Addresses Testability:* Agile champions early test development by practice, and supports testability of requirements and the system throughout development. Agile emphasizes writing tests early in development while developing code, and frequently releasing software that is fully tested[83]. In XP, customers write the tests first for the developers to program to pass these tests.

User stories also provides testability aspects of the system. The user stories offer insight on what needs to be tested. However, because user stories are informal, there is a limitation into what testability aspects they can actually provide and no formal way to assess and test user stories.

*P4: Addresses Validation:* Due to the testability aspects of Agile, validation is also addressed with Agile in that working software already involves a large integration effort.

*P5: Accessibility to Relevant Information:* Almost no formal documentation is produced with Agile development. Relevant “information” is co-located between working code, user stories, and test cases. There are no formal configuration management of user stories in particular, as most of the user stories tend to be written on index cards and posted in a common area for developers to access and implement[25]. The accessibility of pertinent information for development is not formally maintained, and because there is no formal documentation of requirements, no formal capturing of architectural elements or design decisions, the accessibility of relevant information is mostly found in the working code base that all developers likely have access to.

*P6: Impact of Changes:* The Agile community champions “embracing change”, particularly with requirements being introduced or modified later in development. This impact of change is seen as an opportunity in the Agile community and less as a detriment. In government and defense projects, requirements that crop up later in the project is seen as as a risk rather than an opportunity because phases of development are limited by time, resources, and rework is highly undesired. Furthermore, if changes are actually *not* not in the requirements but in the actual design approach to the system, “embracing” this change late in development can hinder the success of the project. The cost of

reworking a number of implementations is high.

The impact of change, with Agile, largely depends on what the change is. Changes in code, this change is seen as an opportunity to improve the integrity of the system. While code changes does affect the requirements of the system, this change is minimal because requirements are not formally captured as part of the development process. Anything more could mean a different picture, particularly in changing a major aspect of the system's architecture. The feature, problem space, or customer test is crucial for development, so a change (always likely) can have a detrimental or beneficial impact[24].

One major cost of Agile is embracing added requirements or requirement changes late in the development process[11]. While the intent of this principle is to keep up with the customer's competitive advantage (as stated in the Manifesto), requirement changes late in the development can be detrimental in large-scale projects (particularly in government contracts for fixed-price with fixed milestones where requirement changes later in the contract may not always be accommodated). In these cases, late requirement changes can be seen as a project hindrance rather than an opportunity. This is most costly when hardware has already been manufactured where adopting new requirements could mean restarting the entire effort which should always be avoided.

*P7: Producing Artifacts:* Producing artifacts include a a working code base, customer tests, test cases to ensure working software.

*P8: Level of Detail:* No process overhead to deal with documentation, writing specifications, or capturing designs in any modeling techniques. Design decisions, if selected as requirements, are implemented through code.

## 6.10 Ad-Hoc Approach

The ad-hoc approach in developing requirements is to not call on any RET to deriving requirements for the system. This approach largely involves numerous meetings with stakeholders and deriving standalone requirements statements with no consideration to the design. The ad-hoc approach might involve a reverse-engineering of requirements from an implemented and working system, where capturing requirements are an afterthought after the system is implemented.

Various ad-hoc approaches are practiced in industry because of number of limitations in project factors such as time and resources available. One limitation is using an ad-hoc approach is that sufficient coordination of critical design decisions will not be expressed in standalone requirements statements. This can lead to poorly specified requirements. A second limitation is that if requirements from a previous system is used, there is no ability to assess the correctness of those requirements. This is a common approach in defense projects in particular; in fact, the reuse of specifications and previous architectures can often be a selling point for a new proposal. Reuse of requirements forces a constraint on new or novel requirements to be developed. Thirdly, there is always the strong likelihood that new personnel turnover will happen for long-term systems. Therefore, previous knowledge might be inaccessible after veteran experts leave the project.

Due to time constraints or project pressures, there might not be the option to develop a new design of a system that has already been built before. It is too easy to just reuse a previous body of knowledge for requirements of a new system so that there can be a cut in time and costs to designing a new system. Previous specifications that were poorly written compound the problem because these specifications do not capture the rationale and reasoning behind the decisions of the system. Therefore, it can be hard to leverage from old specifications. The quality of that previous body of knowledge, however, is generally unknown and the risk is there that it might not be good quality work. This leads to many problems in development where a valid RET approach may improve the situation. An ad-hoc approach is not recommended for design and requirements of a

system.

## 7 Summary Analysis

The major contribution of this survey is an assessment of whether popular RETs support the inclusion of prior design knowledge and how well those RETs support the co-development of design using the evaluation framework presented in Section 5 . The RETs surveyed have characteristics that contribute to design and requirements development that supports system conception. However, all the RETs surveyed still lack necessary support of critical design choices. This section highlights the insights gained from the evaluation of the RETs discussed. These insights provides a basis to tackle future work of this research.

### *P1: Maps to Architecture*

Without a correlation to a given architecture, requirements are hard to articulate. It is difficult to decouple requirements from architecture because architecture provides the blueprint of the system that requirements describe. Several of the RETs surveyed associate an architecture to its context but do not explicitly call out for a pure mapping to a correlating architecture. Problem Frames, Use Cases, and Domain Modeling come closest to satisfying this property.

Problem Frames captures a number of important elements for architecture decisions including behavior, composition, and interfaces between problems. Strong separation of concerns in the problem space and continuous problem decomposition are central tenets of PF. Architectural Frames[80], a specific instance of Problem Frames, translates a chosen architectural style to Problem Frame schematics. Architectural Frames traverses through a similar decomposition as Problem Frames, but in the context of styles, components, and connectors. Architectural Frames drives a solution by placing more emphasis on the chosen architectural style and less on the phenomena of the problems, which is the original basis of Problem Frames.

Use Cases fosters a close mapping to architecture and supports the development of architectural styles, component and connector structure, and implementation such as classes, code structure, and attributes. The accurate correlation to the architecture is dependent on proper decomposition of the use cases themselves.

Domain Modeling emphasizes mapping to architecture exemplified by domain-specific architectures[86] that represent a specific mapping to architecture.

Agile is furthest from mapping to an architecture because it does not place emphasis on architectural decisions to be developed or captured in any manner. In fact, an Agile project's code base is its architecture. As a result, the architecture may constantly evolve and can be lost in the details of the code.

### *P2: Expressibility of Critical Design Choices*

The expressibility of critical design choices is crucial to requirements development because the decisions made about a system's design ultimately drive the requirements of that system. If the requirements do not reflect the design of the system, then there is no basis of rationale and reflection of critical design choices.

Several of the RETs surveyed provide some expressibility of critical design choices. Use Cases and Domain Modeling best exhibit this property among the surveyed RETs. Use Cases capture class structures, component and connector, functions, and behavior. The graphical representation of these choices makes it even more expressive, creating an easier to understand visual association between design decisions and requirements.

Domain Modeling supports expressibility of design choices with the use of domain taxonomies and structures and provides standards for describing characteristics of domains and their properties.

Formal Language and specification is furthest from supporting critical design choices. Design choices such as rationale, structures, and taxonomies are not easily reflected in Formal Languages.

### *P3: Addresses Testability*

Testability of a system is a primary concern of Requirements Engineering [73]. Design also concerns itself with the verification of these decisions[86]. Four of the RETs directly address the testability aspects of requirements capture.

Use Cases provides foundation and a basis for test cases for the system. The approaches taken to test use cases are shaped by the structure of the use cases. Formal Language also supports the testability of requirements by providing an estimate of the environment that the system's requirements must perform to. TDD supports testability in that it places emphasis on writing test cases before development of code. Finally, Agile places great emphasis on frequent releases of fully tested code, which enforces early test development and the testability of the implementation (and its requirements).

#### *P4: Addresses Validation*

No particular RET directly addressed validation of the system.

Use cases gives great insight into the usages of the system which in turn, provides insight into the validation of the system. The collection of the use cases gives an informal assessment of the entire system.

TDD and Agile RETs also offer insight to the validation of the whole system because of the emphasis of upfront and continuous testing. Though TDD and Agile also do not explicitly call for validation, the continuous integration efforts taken to test the system provides insight to validation of the entire system.

#### *P5: Accessibility to Relevant Information*

With the exception of Formal Languages, all RETs support accessibility of relevant information, while use cases most closely support this property. Use cases capture a sufficient amount of relevant information that usually requires accessibility for other associated use cases. Use cases have the capability to make accessible structures public and visible to other associated use cases. Therefore the accessibility to critical data is entirely possible.

Formal Language offers the least support for this property because a formal specification is generally confined to one artifact or document, and does not call for association to other pieces of information. The focus of Formal languages and specifications is on the preciseness and correctness of system properties.

#### *P6: Impact of Changes*

With the exception of Formal Languages, all RETs support efficient handling of impacts of changes made in relevant information. Use cases and Agile meet this property the most.

A change in relevant information in a use case can be realized much sooner, because there is direct linkage between use cases. Therefore if there are interdependencies in data between associated use cases, a change in one piece of data from one use case will be evident in the system. Use cases are co-located in one environment which minimizes the impact of a change.

Agile encourages “embracing change” even when requirements are introduced later in the development process[11]. This principle re-orientes Agile development to essentially be prepared for changes at any given time; preparation for change is built into the development process of the project. The impact of changes in Agile development is not intended to be major re-work of the code base, but rather to adapt to the changes as they come. Agile supports this property fairly well.

#### *P7: Producing Artifacts*

All the evaluated RETs produce a variety of different artifacts to capture information. It is not applicable to assess which RET met this property better than others.

What is interesting to note about artifacts that each RET can produce is that there is no single consistent manner to capture and express requirements. Requirement information is captured using a number of methods such as charts, diagrams, symbols, boxes and arrows, textual annotations,

notations and mathematical formulas, code structure, specifications, and test cases. The variety of artifacts that can be produced by these different RETs makes it difficult to assess what is appropriate for the product. These artifacts make it difficult to trace information from requirement to architecture or requirement to code when the requirements themselves are captured inconsistently.

#### *P8: Level of Detail*

All the RETs evaluated capture information at various levels of detail. It is not applicable to assess which RET met this property better than others.

There are two issues to consider, however, for this property. One is the issue of what type of detail is required for each RET so that the detail is sufficient in understanding requirements. The other issue is the amount of detail to be captured and whether the more fine grained the detail, the easier it is to produce requirements. For instance, Problem Frames captures an abstract level of detail that might help give context to the system's concept, but still requires significant decomposition of the sub-problems and solutions to these sub-problems. The details captured need to be relevant and profound, but this might take several iterations till the level of detail is entirely useful.

Use cases, on the contrary, provides the options of granularity of details in the use case. An informal use case can provide very little information that will better develop requirements, whereas a formal use case will contain several layers of critical details. Use cases will capture usages of the system (pre and post conditions, triggers, as examples) and will also capture information for code structure (ports, connectors, variables, class structure). This level of detail gives basis to testing and development.

#### *Summary*

The major contribution of this survey is that through the evaluation framework presented, an assessment can be made about whether the RETs supports the inclusion of design knowledge and how well the RETs supports the co-development of design. The RETs surveyed have positive aspects that contribute to design and requirements development and are applicable. However, all the RETs surveyed still have problematic areas with supporting critical design choices.

## **7.1 Evaluation Framework Limitations**

The evaluation framework presented does have limitations, discussed in this section.

One major limitation of this framework is that the assessment of a particular RET to the framework is entirely variable on the objective of the RET. The objectives of Formal Languages, for example, are widely different than the objectives of Agile Development. Therefore, an RET is likely to not explicitly meet properties that are more appropriate for one RET over another.

A recovery from this limitation is that it forces more partitioning of the categories that are applicable for a basis of the future of this work. This work already narrowed down the main categories of RETs to look at RETs and its inclusion of design. For future work, we can refine the scope again, and specifically look at a smaller number of RETs that address the inclusion of design concepts already and focus on gaps in those RETs.

Furthermore, because it is not realistic that one RET can meet all eight properties of the framework sufficiently, it will force this work to select which of the properties are important for the future of this research. However, for completeness, it was important and valuable to evaluate all the RETs against the other properties that also support design decisions made of the system.

## 8 Survey Insights

The evaluation of the nine RETs surveyed provided five key insights to requirements development coupled with design, and employing these RETs in practice.

### *Insight 1*

Contextual information - or metadata - is needed in order to develop requirements for understandability of a system. Requirements metadata is dispersed in a variety of places across a project: the architecture and its definitions, verification approaches of the system, the rationale of key decisions made about a system, behavioral constraints, domain terminology, as examples. These pieces of information have a great influence on the actual requirements and expectations of the system.

Requirements development, like architecture, is an expression of the critical design decisions made about a system. Therefore the representations of these design choices are vital to understand the problem the system addresses. Since each RET has a unique type of output that only expresses unique pieces of information required for each respective RET, there is more information that is missing and required to comprehend the system. The surveyed RETs provide outputs in different formats: standalone textual statements (Ad-Hoc), symbol and formulas (Formal Language), or pictorial representations (use cases, goal models) of the system. The outputs of each of the surveyed RETs are simply not rich enough to fully analyze requirements and ensure completeness of the requirements of a system.

### *Insight 2*

Each RET has different objectives and unique benefits when called upon, therefore it is awkward and unlikely for all RETs to be called upon in a given project. Each RET supports project needs using an exclusive approach that will be complimentary to the project. There are RETs that are also conflicting and counterproductive for particular types of projects making it awkward for all RETs to be utilized in one project.

Multiple RET usage in one project sitting also encourages an inconsistency of development, making it difficult to assess accuracy of development information. Such inconsistencies make measuring progress cumbersome. RETs require different levels of granularity in details and information. Because no particular RET comes with standards to measure correctness of the information collected, an aggregate usage of all the RETs can give an inconsistent picture of the system. Notations and artifacts of each RET are vastly different; tracing information between different notations and artifacts hinders progress. The value of performing this reconciliation is small, and the mis-interpretations between such different outputs that each of these RETs offer are too great.

Moreover, switching from one RET to another in one project sitting is costly. There is an adjustment that is required from startling switches, especially if the two RETs in question are not complimentary. This adjustment is costly in time and resources, as opposite RETs often have gaps in information that is needed for another RET. For example, moving from Problem Frames to Agile is difficult because Problem Frames do not encourage the development of a code base, continuous integration and updating of code, and the development of user stories. Instead, Problem Frames focuses on decomposing the larger system objective to manageable and smaller problems, and analyzing constrains and expected behaviors of these smaller problems. Agile focuses on user stories, customer input, and continuous development of a code base where concepts are continuously changing. Agile encourages a re-visit of concepts, code, and customer acceptance whereas Problem Frames establish context and expectations of the system (and code follows much later in development). A move from one RET to another opposing RET requires a learning adjustment and time to populate gaps of information that can come with drastic switches.

As another example, TDD focuses on the testability aspect of the system upfront, and not heeding to architectural blueprints and design decisions. TDD does not encourage formal architectural decisions to be made early in system development; rather, TDD focuses on testability of potential

requirements. In systems where software significantly drives the system and many unknowns may exist, the testability aspect of the system will be a significant driver to requirements synthesis.

Agile examines user stories for requirements development. User stories capture a variety of pieces of information; therefore there is no standard to derive requirements and architectural decisions of the system. Such an approach works well in working environments where the project is largely driven by the user’s needs of the system for input and feedback. User needs are an aspect of critical decisions, constraints on these needs may be imposed by limitations of the implementation or by business decisions and priorities of features to be implemented. There are no standards to assess the correctness or completeness of user stories. In Agile, critical information can be mostly found in user stories, which are not very reliable for understanding a complete system’s implementation and design. Agile focuses on the user needs and aspects with no formal architectural guidance or captured and analyzed decisions.

There are pairs of RETs that are diametric opposites in terms of approach and objective. Formal Language and Agile are great examples of this, where Formal Language focuses on preciseness of requirements while Agile focuses on user needs.

Since each RET offers different benefits, each RET can also offer disadvantages to particular types of projects. For instance, a web-based product that is largely driven by user queries and relies on visually pleasing elements on the screen to keep a user’s interest (such as a banking system), will unlikely capture their requirements in a Formal Language or syntax. Such mathematical precision would be least productive for a web-based product whose main needs will be to engage the user’s interest.

While there are pairs of RETs that are complimentary to use on a project (Insight 3), there is still an awkwardness in calling all RETs in one project.

*Insight 3*

RETs tackle different software development problems and different phases of software development as shown in Figure 7. The use of one RET at one time phase of the project could ultimately lead to the use of another RET at a different phase of the project. It is highly likely a project will call upon more than one RET in a project’s duration.

For example, large-scale development tends to start on early prototypes to understand the system’s needs. These early prototypes, for instance, give a sense of possibilities for requirements, in seeing how modules interact together. Early prototypes can often lead to distinct use cases and possible standalone statements to be derived. Early prototypes can also be the basis of early testing possibilities (TDD). This combination is not uncommon in practice[26].

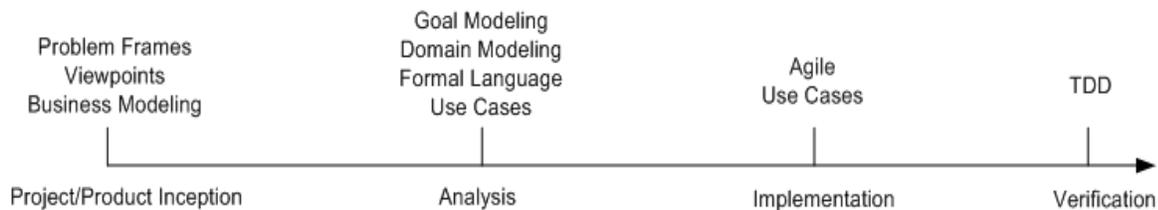


Figure 7: RETs tackle different software development problems

*Insight 4*

Several of the RETs surveyed call upon solution-driven techniques to create requirements and understand the problem being decomposed.

A number of the RETs surveyed foster resulting solution artifacts (code base, test cases, as examples) to derive the requirements of the system. Agile approaches, for instance, encourage early code base development and continuous feedback of features that have been implemented. By enforcing a working product at every iteration [reference Agile Manifesto], there is always

a solution to potential features because these potential features are implemented and will work. Testing aspects of the system is another solution-driven approach to understanding requirements of a system. TDD encourages early test cases and running code, therefore partial solutions to the system are already developed and tested.

RETs fundamentally derive solutions first: early testing, code development, user stories; before deriving the requirements of the system. The results of these early solution efforts provide rationale to the requirements derived.

#### *Insight 5*

Inclusion of critical design knowledge fosters better decision making of requirements. Product-line systems tend to look at previous history and domain knowledge to get started in development. This history is not limited to a previous codebase for reuse, domain knowledge and terminologies, or even previous requirements.

Therefore separating requirements and design is inherently difficult in large-scale projects. Potential solutions heavily influence the way we describe the problem. With domain specific architectures, there are already a number of domain-driven requirements that exist for systems in specific domains.

## **8.1 Future Work**

The biggest distinction between Requirements Engineering and other software and system development activities is that Requirements Engineering is the only activity that focuses on the problem space while all the other activities (architecture, code, testing) focus on the solution space of the system[21].

In practice, software requirements development has largely been isolated from other software activities, bringing forth a number of problem areas in the field. This survey analyzed nine accepted RETs that are widely practiced in industry and studied in academia, demonstrating that these RETs are still missing key solutions to common requirement problems of incompleteness and analyzability. These nine RETs were evaluated using eight exclusive properties that examined the RETs abilities to address areas of concern in architecture, detail, testing, and design expressivity. This survey demonstrated that requirements and design development are significantly intertwined.

The future of this research focuses on the key insights from this survey and how to apply these insights to software requirements development. This research will use the Twin Peaks paradigm as a consistent model, where the focus will be on the support of the intertwining with requirements activities and inclusion of design details. Currently, there is no technique that directly supports these two activities in parallel specifically.

## 9 References

### References

- [1] Military standard 498. U.S. Department of Defense Directive, May 12 2003.
- [2] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14(2):159–192, 1982.
- [3] Hans Akkermans and Jaap Gordij. What is this science called requirements engineering? In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference*, pages 266–271, Washington, DC, USA, 2006. IEEE Computer Society.
- [4] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.
- [5] Kenneth M. Anderson, Susanne A. Sherba, and William V. Lephien. Towards large-scale information integration. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 524–534, New York, NY, USA, 2002. ACM.
- [6] Paul Arkley and Steve Riddle. Overcoming the traceability benefit problem. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 385–389, Washington, DC, USA, 2005. IEEE Computer Society.
- [7] Paul Arkley, Steve Riddle, and Tom Brookes. Tailoring traceability information to business needs. In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference*, pages 234–239, Washington, DC, USA, 2006. IEEE Computer Society.
- [8] Debra Aubrey. Controlling the hms program through managing requirements. In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference*, pages 222–227, Washington, DC, USA, 2006. IEEE Computer Society.
- [9] Kent Beck. *Test-Driven Development By Example*. Addison Wesley, 2004.
- [10] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.
- [11] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. Web page. Accessed March 8, 2009.
- [12] George Boby and Laurie Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337–342, April 2004.
- [13] Barry Boehm. Get ready for agile methods, with care. *Computer*, 35(1):64–69, 2002.
- [14] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.
- [15] Davide Bolchini, Franca Garzotto, and Paolo Paolini. Branding and communication goals for content-intensive interactive applications. *Requirements Engineering, IEEE International Conference on*, 0:173–182, 2007.
- [16] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

- [17] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12:34–41, 1995.
- [18] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.
- [19] Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [20] R. J. A. Buhr. Use case maps as architectural entities for complex systems. *IEEE Trans. Softw. Eng.*, 24(12):1131–1155, 1998.
- [21] Betty H. C. Cheng and Joanne M. Atlee. Research directions in requirements engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 285–303, Washington, DC, USA, 2007. IEEE Computer Society.
- [22] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.
- [23] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [24] Alistair Cockburn. *Agile software development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [25] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.
- [26] Leyna Cotran, Susan Sim, and John Noll. Adding Agile to an Embedded Systems project. North Carolina State University Department of Computer Science’s Inaugural Symposium for Graduate Students, March 2008.
- [27] Daniela E. Damian and Didar Zowghi. The impact of stakeholders? geographical distribution on managing requirements in a multi-site organization. In *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 319–330, Washington, DC, USA, 2002. IEEE Computer Society.
- [28] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering. *Software Engineering, International Conference on*, 0:612, 1997.
- [29] Steve Easterbrook. Resolving requirements conflicts with computer-supported negotiation. pages 41–65, 1994.
- [30] Steve Easterbrook, Robyn Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Trans. Softw. Eng.*, 24(1):4–14, 1998.
- [31] Steve Easterbrook, Eric Yu, Jorge Aranda, Yuntian Fan, Jennifer Horkoff, Marcel Leica, and Rifat Abdul Qadir. Do viewpoints lead to better conceptual models? an exploratory case study. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 199–208, Washington, DC, USA, 2005. IEEE Computer Society.
- [32] Alexander Egyed, Paul Grnbacher, and Nenad Medvidovic. Refinement and evolution issues in bridging requirements and architecture - the cbasp approach. In *The First International Workshop on From Software Requirements to Architectures (STRAW01)*, pages 42–47, 2001.

- [33] Richard E. Fairley and Richard H. Thayer. The concept of operations: The bridge from operational requirements to technical specifications. *Ann. Softw. Eng.*, 3:417–432, 1997.
- [34] S. Faulk, L. Finneran, J. Jr. Kirby, S. Shah, and J. Sutton. Experience applying the core method to the lockheed c-130 software requirements. In *Ninth Annual Conference on Safety, Reliability, Fault Tolerance, Concurrency, and Real Time Security*, pages 3–8, 1994.
- [35] K Forsberg and H Mooz. The relationship of systems engineering to the project cycle. *Engineering Management Journal*, 4(3):36–38, 1992.
- [36] M. Fowler. “Use and Abuse Cases”. *Distributed Computing*, April 1998.
- [37] Joseph A. Goguen. Requirements engineering as the reconciliation of technical and social issues. In *in Requirements Engineering: Social and Technical Issues*, pages 165–199. Academic Press, 1994.
- [38] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101, 1994.
- [39] B. Hailpern and P. Tarr. Model-driven development: the good, the bad, and the ugly. *IBM Syst. J.*, 45(3):451–461, 2006.
- [40] Anthony Hall. Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19, 1990.
- [41] Anthony Hall. Using formal methods to develop an atc information system. *IEEE Softw.*, 13(2):66–76, 1996.
- [42] Jon G. Hall, Michael Jackson, Robin C. Laney, Bashar Nuseibeh, and Lucia Rapanotti. Relating software requirements and architectures using problem frames. In *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 137–144, Washington, DC, USA, 2002. IEEE Computer Society.
- [43] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Trans. Softw. Eng.*, 22(6):363–377, 1996.
- [44] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Softw. Eng.*, 6(1):2–13, 1980.
- [45] J. Hughes, J. O’Brien, T. Rodden, M. Rouncefield, and I. Sommerville. Presenting ethnography in the requirements process. In *RE '95: Proceedings of the Second IEEE International Symposium on Requirements Engineering*, page 27, Washington, DC, USA, 1995. IEEE Computer Society.
- [46] IEEE. *IEEE Recommended Practice for Software Requirements Specifications*, ieee std-830 edition, 1998.
- [47] Daniel Jackson. Lightweight formal methods. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, page 1, London, UK, 2001. Springer-Verlag.
- [48] M. Jackson. Problems and requirements [software development]. In *RE '95: Proceedings of the Second IEEE International Symposium on Requirements Engineering*, page 2, Washington, DC, USA, 1995. IEEE Computer Society.
- [49] M Jackson. Problem frames and software engineering. *Information and Software Technology*, 47(14):903–912, 2005.

- [50] Michael Jackson. *Software requirements & specifications: a lexicon of practice, principles and prejudices*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.
- [51] Michael Jackson. Why software writing is difficult and will remain so. *Inf. Process. Lett.*, 88(1-2):13–25, 2003.
- [52] Michael Jackson. The problem frames approach to software engineering. In *APSEC*, page 14, 2007.
- [53] Michael Jackson and Paula Zave. Domain Descriptions. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, pages 56–64, Los Alamitos, CA, 1993. IEEE Computer Society.
- [54] Sami Jantunen, Kari Smolander, and Donald C. Gause. How internationalization of a product changes requirements engineering activities: An exploratory study. *Requirements Engineering, IEEE International Conference on*, 0:163–172, 2007.
- [55] David Janzen and Hossein Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.
- [56] David S. Janzen and Hossein Saiedian. On the influence of test-driven development on software design. In *CSEET '06: Proceedings of the 19th Conference on Software Engineering Education & Training*, pages 141–148, Washington, DC, USA, 2006. IEEE Computer Society.
- [57] Ivan J. Jureta, Stéphane Faulkner, and Pierre-Yves Schobbens. Justifying goal models. *Requirements Engineering, IEEE International Conference on*, 0:119–128, 2006.
- [58] Haruhiko Kaiya and Motoshi Saeki. Using domain ontology as domain knowledge for requirements elicitation. In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference*, pages 186–195, Washington, DC, USA, 2006. IEEE Computer Society.
- [59] Benjamin L. Kovitz. *Practical software requirements: a manual of content and style*. Manning Publications Co., Greenwich, CT, USA, 1999.
- [60] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE '01: Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, page 249, Washington, DC, USA, 2001. IEEE Computer Society.
- [61] Robin Laney, Leonor Barroca, Michael Jackson, and Bashar Nuseibeh. Composing requirements using problem frames. In *RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International*, pages 122–131, Washington, DC, USA, 2004. IEEE Computer Society.
- [62] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, 2003.
- [63] Sotirios Liaskos, Alexei, Yijun Yu, Eric Yu, and John Mylopoulos. On goal-based variability acquisition and analysis. *Requirements Engineering, IEEE International Conference on*, 0:79–88, 2006.
- [64] Susan Lilly. Use case pitfalls: Top 10 problems from real projects using use cases. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 174, Washington, DC, USA, 1999. IEEE Computer Society.
- [65] E. Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 564–569, Washington, DC, USA, 2003. IEEE Computer Society.

- [66] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. Moving architectural description from under the technology lamppost. *Inf. Softw. Technol.*, 49(1):12–31, 2007.
- [67] Tim Menzies, Steve M. Easterbrook, Bashar Nuseibeh, and Sam Waugh. An empirical investigation of multiple viewpoint reasoning in requirements engineering. In *RE '99: Proceedings of the 4th IEEE International Symposium on Requirements Engineering*, page 100, Washington, DC, USA, 1999. IEEE Computer Society.
- [68] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 243–253, 2007.
- [69] Steven P. Miller, Alan C. Tribble, Michael W. Whalen, and Mats P. E. Heimdahl. Proving the shalls: Early validation of requirements through formal methods. *Int. J. Softw. Tools Technol. Transf.*, 8(4):303–319, 2006.
- [70] Clementine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jezequel. Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering*, 32(3):140–155, 2006.
- [71] Bashar Nuseibeh. Towards a framework for managing inconsistency between multiple views. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 184–186, New York, NY, USA, 1996. ACM.
- [72] Bashar Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–117, 2001.
- [73] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, New York, NY, USA, 2000. ACM.
- [74] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. Viewpoints: meaningful relationships are difficult! In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 676–681, Washington, DC, USA, 2003. IEEE Computer Society.
- [75] Ken Orr. Agile requirements: Opportunity or oxymoron? *IEEE Softw.*, 21(3):71–73, 2004.
- [76] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Sci. Comput. Program.*, 25(1):41–61, 1995.
- [77] Mario Pichler, Hildegard Rumetshofer, and Wilhelm Wahler. Agile requirements engineering for a social insurance for occupational risks organization: A case study. In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference*, pages 246–251, Washington, DC, USA, 2006. IEEE Computer Society.
- [78] Klaus Pohl and Ernst Sikora. Cosmod-re: Supporting the co-design of requirements and architectural artifacts. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 258–261, 2007.
- [79] Colin Potts. Re-framing requirements engineering. In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference*, pages 278–283, Washington, DC, USA, 2006. IEEE Computer Society.

- [80] Lucia Rapanotti, Jon G. Hall, Michael Jackson, and Bashar Nuseibeh. Architecture-driven problem decomposition. In *RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International*, pages 80–89, Washington, DC, USA, 2004. IEEE Computer Society.
- [81] W Royce. Managing the development of large software systems. In *IEEE WESCON*, pages 1–9. IEEE, Aug. 1970.
- [82] Mohammed Salifu, Yu Yijun, and Bashar Nuseibeh. Specifying monitoring and switching problems in context. In *RE '07: Proceedings of the 15th IEEE International Requirements Engineering Conference*, 2007.
- [83] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
- [84] Ian Sommerville, Peter Sawyer, and Stephen Viller. Viewpoints for requirements elicitation: A practical approach. In *ICRE '98: Proceedings of the 3rd International Conference on Requirements Engineering*, pages 74–81, Washington, DC, USA, 1998. IEEE Computer Society.
- [85] William Swartout and Robert Balzer. On the inevitable intertwining of specification and implementation. *Commun. ACM*, 25(7):438–440, 1982.
- [86] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architectures: Foundations, Theory, and Practice*. John Wiley & Sons, 2008.
- [87] Telelogic. Telelogic DOORS: Requirements Management for Complex Systems and Software Development. Web page. Accessed April 4, 2009.
- [88] Telelogic. Telelogic Tau: Model Driven Development of Complex Systems and Services. Web page. Accessed April 4, 2009.
- [89] F. Torner, M. Ivarsson, and F. Pattersson. An empirical quality assessment of automotive use cases. In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference*, pages 86–95, Washington, DC, USA, 2006. IEEE Computer Society.
- [90] Daniel Turk, Rober France, and Bernhard Rumpe. Assumptions Underlying Agile Software-Development Processes. *Journal of Database Management*, 16(4):62–87, Oct-Dec 2005.
- [91] Damien Vanderveken, Dewayne Perry, and Christophe Ponsard. Deriving architectural descriptions from goal-oriented models. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*. ACM, 2006.
- [92] Stephen Viller and Ian Sommerville. Social analysis in the requirements engineering process: From ethnography to method. In *RE '99: Proceedings of the 4th IEEE International Symposium on Requirements Engineering*, pages 6–13, Washington, DC, USA, 1999. IEEE Computer Society.
- [93] Colin Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers, 2nd edition, 2004.
- [94] Anthony I. Wasserman, Robert J. Muller, and Peter A. Pircher. The object-oriented structured design notation for software design representation. *Computer*, 23(3):50–63, 1990.
- [95] Tim Weilkiens. *Systems Engineering with SysML/UML*. Morgan Kaufmann, San Francisco, Calif, USA, 2008.
- [96] Jim Whitehead. Collaboration in software engineering: A roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 214–225, Washington, DC, USA, 2007. IEEE Computer Society.

- [97] Jon Whittle and Praveen K. Jayaraman. Generating hierarchical state machines from use case charts. In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference*, pages 16–25, Washington, DC, USA, 2006. IEEE Computer Society.
- [98] Jeannette M. Wing. A specifier’s introduction to formal methods. *Computer*, 23(9):8–23, 1990.
- [99] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, 1997.

Appendix A: RET Placemat

Evaluation	Problem Frames	Viewpoints	Use Cases	Formal Language	Test-Driven Approaches	Domain Modeling	Business Modeling	Goal Oriented Approaches	Agile
<b>P1 Maps to Architecture</b>	Forces a separation of concerns between the machine and the software that executes on that machine. [79]: Architectural Frames translates the chosen architectural style and translates to Problem Frame schematics.	Separation of concerns; examines alternative representations of the system based on different views of the system. [80]: Complexity was a big issue in this case study, re-emphasizing the needs to map architectural models with uses of the system to understand what the models mean	If properly decomposed and modeled, use cases can map appropriately to architecture. [126]: Use Case Maps are presented as "architectural entities" in order to gain insight to all developmental phases of the system (UCMs are not recommended).	Formal language can be used for architectural models through type-checking the implementation of architectural components [85].	As the code base and test cases grow in the system, the architecture can become less understood or lost in the code base.	Captures key information that influences an architecture: properties, execution on an application domain; domain-specific software architectures (DSSAs) [85].	A strong architecture can greatly influence the business model. The heart of a great product is with a good architecture [85].	Goal models capture expected interfaces between a system and external factors which can also be reflected in an architecture. [90] Uses a component and connector description language, goals are used to derive the behaviors of the architecture	No formal specification or design required to be captured, no mapping of code to an architecture, the code is the architecture.
<b>P2 Expressibility of Critical Design Choices</b>	After further decomposition of the larger problem, sub-problems can be addressed with regard to design. [61]: Composition Frames emerges inconsistencies in solution space and weighs potential design solution options.	Different and conflicting viewpoints are design drivers to the system; information captured in views provide design context and areas of exploration. [32]: Case study where data site explored tolerance in inconsistencies and its affect on the system's design. [80]: issues of scalability in this case study; emphasizing the need to address design for scalability issues.	Adheres to design elements of a system: structures, functions, classes, components, connectors.	Expresses the system behaviors in terms of functionality, timing, performance, and internal structure. Design decisions such as rationale, component definition, and styles are not easily reflected.	the design can be lost in the code base because of test development is more emphasized than upfront design planning	Expressibility in taxonomies and structures; provides standards for describing characteristics and nomenclatures.	Requirements are many times driven by business requirements and design choices. A user interface can be greatly influenced by the business model and goals [16].	Goals encapsulates a number of design choices that can include interactions with humans and users, devices, and the environment that the system must be compatible with. Goals can also include the non-functional or usage properties, where multiple design elements for instance, must interplay together to meet a system goal.	Supports some expressibility of critical design choices made through working case base, continuous testing of code, addressing stakeholder concerns (customer, user). Does not call for specific architectural and design tasks; design is assumed to be part of code base

Appendix A: RET Placemat

Appendix A: RET Placemat

Evaluation	Problem Frames	Viewpoints	Use Cases	Formal Language	Test-Driven Approaches	Domain Modeling	Business Modeling	Goal Oriented Approaches	Agile
<b>P3</b> Addresses Testability	Not applicable	Not applicable	Shapes the test cases and test approaches of the system; basis of writing test cases.	Not a replacement of testing activities [41]; supports testability in providing an approximation of the real environment the system must function in.	TDD facilitates testing activities.	Not applicable	Not applicable	Not applicable	Early test development and frequently releasing software that is fully tested[82]. XP emphasizes customers written tests
<b>P4</b> Addresses Validation	Not applicable	Not applicable	Not directly applicable. Collective use cases implies some validation of the system.	Offers insight into the validity of the design approach through the emphasis of proof of correctness of system properties.	Continuous testing of code base gives some insight to the validation of the system.	Not applicable	Not applicable	Not applicable	Due to the testability aspects of Agile, much of the validation is also addressed in that working software already involves a large integration effort.
<b>P5</b> Accessibility to Relevant Information	Relevant information is captured in the Problem Frame and easily accessible. [61]: Composition Frames carries multiple design solutions for accessibility and assessment.	Relevant information is captured in the Viewpoint. [29] Synoptic tool supports Viewpoints in one environments, making access to conflict information easier	Use cases are accessible in one environment; captures critical architectural information of an actor or an interface for example.	Not applicable	Relevant information found in the test cases themselves.	Captures relevant information pertinent to a domain's characteristics; easily accessible; provides a rationale basis for future decisions in design.	Not applicable	Not difficult to access relevant information if goals are captured in one environment or specification	Relevant information is co-located between working code, user stories, and test cases
<b>P6</b> Impact of Changes	The impact of changes made in a Problem Frame does affect the overall design schema, particularly where design drivers can be affected. [59] suggests the use of domain ontology.	Change occurs when conflict information/views are resolved. Impact can be detrimental if not resolved, or ensure progress when resolved.	changes realized sooner in development; direct linkage to associated use cases where changes can be realized quicker.	Not applicable	Promotes accurate and up to date test cases so the impact of a change is kept up to date.	If meaning and intent of a domain is altered, the impact of these changes can be quite costly to rework of architectures and requirements.	A change in the business model can affect the implementation of the systems that support the business.	A change in a goal can trigger a change in concept and design.	"embracing change", particularly with requirements to be introduced late in development[11].
<b>P7</b> Ties to System Concept	Decomposed problems capture conceptual information tied to a system concept.	Viewpoints can contain information that is conflicting and inconsistent, therefore maintaining a tie to the system concept can be variable.	The collective associated use cases maintains the system's concept. [96]: a synthesis of requirements UML models supports development of state behavior of a system.	Formal notation used to describe and measure a system property is directly linked to the system concept when seeking to prove the correctness of design implementations and choices.	test cases can be categorized by the functions they test; however the "big picture" concept can be missed.	The domain model is directly tied with the system's concept in that it provides guidance and context to the system structure[85].	Describes the business environment that the system will support. If the business need changes, this can directly affect the implementation of the system.	Goal modeling fosters reasoning of the system's objectives and manifests the rationale behind a system concept.	System concept is found in the working code

Appendix A: RET Placemat

Appendix A: RET Placemat

Evaluation	Problem Frames	Viewpoints	Use Cases	Formal Language	Test-Driven Approaches	Domain Modeling	Business Modeling	Goal Oriented Approaches	Agile
<b>P8 Producible Artifacts</b>	Frames, semantics include state behavior symbology, specifications translated from frames [61].	Conceptual models (informal) [31]; design or architectural descriptions	Use Cases, scenario diagrams, requirement specifications, sequence and activity flow charts; state machine diagrams [86, 87]; traceability matrix between use cases and requirement	Formal specifications with specific language syntax; formal methods for architectural models.	test cases, code base	Models (UML) , DSSAs, domain dictionaries	Business use cases	Goal models and goal use cases; requirements specifications derived from goal models.	A system, user stories, implemented features, customer tests, working software increments
<b>P9 Level of Detail</b>	Capture details in domain characteristics, application domain, interactions. Granularity of details vary.	Granularity in details vary. No standards exist to capturing details though a set of views should capture standard pieces of information. [85]; Should provide filtered information that affects design.	Significant amount of detail captured relating to interactions between actors, connectors, expected usages of the system. [23]; informal, casual, and fully dressed use cases	Quantitative details in proof of correctness of requirements. [85] 3 categories: informal, semiformal, and formal models	Details embedded in test cases but are focused on the testable aspects of the system; design and architectural detail not nominally captured in test cases	Domain models can vary in abstract detail to concrete detail.	Abstract detail; sets context of product of systems supporting a business	Varies: can be abstract, more contextual, or detailed.	No process overhead to deal with documentation, writing specifications, or capturing designs in any modeling techniques. Design decisions are implemented in code.

Appendix A: RET Placemat



Appendix B: Summary Table

	Problem Frames	Viewpoints	Use Cases	Formal Language	Test-Driven Approaches	Domain Modeling	Business Modeling	Goal Oriented Approaches	Agile
<b>Definition/Description</b>	Problem Frames are the structuring of real problems as a collection of interacting subproblems. Each subproblem is a smaller and simpler problem than the original. Each subproblem contains clear and understandable interactions [51, 53].	Capturing separate descriptions of the viewpoints of different stakeholders, identifying and resolving conflicts between them [66]	A description of a software system's properties and behaviors as it reacts to requests external to the system.	A formal specification language provides the notation (syntax) and the domain (semantics) as well as which objects satisfy the specification [98]	Test case development first before production code development [97]. Looks at ways to pass tests and incrementally add functionality, affecting the design.	Looking at the characteristics of a domain and modeling how this domain can interact with other domains. [51]	Requirements are many times driven by business needs. Modeling business needs determines its impacts in areas of user experience, product-line development, and development processes [15]	Goal-oriented requirement approaches capture the objectives of the system at different levels of detail [60].	Champions for early testing, continuous customer feedback, and frequent releases of software [11]
<b>Visualization</b>	[78] RE as a problem of informational display.	glBIS is a graphical tool that represents argumentation process in a hypertext graph. Arguments are linked that either support or refute the position.	Basis of use cases are visual graphical usage and representations. Use cases gets away from the formalities of spec writing and graphical displays potential requirements (though derived). Perspectives can be graphically represented.	Mostly mathematical and textually annotations; though there are tools to support processing of algorithms; other examples such as MATLAB and Simulink are mathematically based with formal methods for implementation		graphical, pictorial	graphical, pictorial	Stated goals, modeling with UML, textual annotations and linkages.	No associated visualization tools; modeling tools for Agile are whiteboards and markers, user story boards [24]
<b>Tool/Tool Kit</b>	Visio, Ppt	Synoptic: tool support for collaborative task-focussed negotiation. glBIS: representation of identified issues and their position of resolution.	DOORS/Tau, Rational Rose, Rhapsody			Visio, Ppt	extensions of UML, SysML; Powerpoint		wikis; FitNesse: tests are expressed as tables of input data and expected output data.
<b>Method/Approach</b>		Context-switching, variability [67], [81]; Viewpoints Framework [70, 66]; PREview [83]; CORE [83]	RUP		early test case development per code base.	DODAF, CBSP [27];	AWARE+ [15], [55]	GAM [58], RSML [49]	user stories, customer feedback and interaction, feature estimation, sprints, pair programming
<b>Notation/Language (Formal/Informal)</b>	problem context symbology [43]; context diagrams [79], domain sets [61];		[96], i*, UML, SysML	RSML (Leveson, UCI) theorem proofs [68]		SysML		GoalML (UCI), KAOS [28], CTL [49]	No formal specification or requirements document called for.

Appendix B: Summary Table

