TECHNICAL DEBT MANAGEMENT IN RELEASE PLANNING – A DECISION SUPPORT FRAMEWORK

by

Isaac Daniel Griffith

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

August 2014

DEDICATION

This thesis is dedicated to my father, Gordon Charles Griffith. He is the person who initially sparked my interest in Logic, Computer Science, and (unwittingly) Software Engineering. It was through his guidance that I wrote my first programs. It was also my father who made me want to dedicate my life to understanding and making software engineering practice better and easier for others.

ACKNOWLEDGEMENTS

When I began this work 3 years ago I was more interested in understanding the world through the eyes of philosophy instead of the eyes of science. At the time, I was eager to develop tools rather than conduct research and had a hard time understanding what science meant in the context of software engineering. I would like to thank my advisor Clemente Izurieta whose wisdom and guidance have enlightened me to a better understanding of research and science. It was through his subtle suggestions and high expectations that I was able to complete this thesis. He provided a continual reminder that one must strive to find a balance in life which allows one to center themselves within a world of ever more intriguing question and problems.

I am also grateful for the members of the Software Engineering Laboratory as well as other fellow graduate students who have, perhaps unbeknownst to them, pushed me to exceed my own boundaries of knowledge and understanding.

Finally, I am most grateful to my parents Gordon and Connie Griffith, who have always supported me and my goals. I would like to thank the most caring and wonderful woman in the world, Lora Whitmore, who has supported me in both my daily life and academic pursuits. Finally, I would like to thank my daughter, Dakotah Clausen, who unbeknownst to her, who has provided an incredible drive to complete my work and better myself every day.

TABLE OF CONTENTS

TABLE OF CONTENTS - CONTINUED

TABLE OF CONTENTS - CONTINUED

TABLE OF CONTENTS - CONTINUED

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

LIST OF FIGURES - CONTINUED

Figure Page

LIST OF FIGURES - CONTINUED

Figure Page

NOMENCLATURE

Developer – Software Engineer
ETD – Effective Technical Debt
FCFS – First Come First Serve
ORP – Operational Release Planning
PTD – Potential Technical Debt
RP – Release Planning
RRP – Release Re-Planning
SEDS – Software Engineering Decision Support
SRP – Strategic Release Planning
TDM – Technical Debt Management
TDMF – Technical Debt Management Framework

ABSTRACT

Technical debt is a financial metaphor used to describe the tradeoff between the short term benefit of taking a shortcut during the design or implementation phase of a software product (e.g., in order to meet a deadline) and the long term consequences of taking said shortcut, which may affect the quality of the software product. Recently, academics and industry practitioners have offered several models and methods which purport to explain or manage this phenomenon. Unfortunately, to date, there has yet to be a framework which supports managers in making decisions regarding technical debt. Although similar solutions exist to support the release planning phase of software development, they focus on the management of new features and do not take into account issues relating to technical debt and its effects on the development process.

This thesis describes a software engineering decision support system focusing on three key components: *analysis and decision, intelligence,* and *simulation*. Supporting each of these components is a meta-model which bridges the gap between technical debt management and software release planning. To investigate the development of the *analysis and decision* and *intelligence* components we used a reduced form of this meta-model in conjunction with a coalition formation games approach. This approach served to evaluate the technical debt management and release planning issues, and was found superior, using simulation, in comparison to a first-come, first-served method (representative of typical agile planning processes). To investigate the development of the *simulation* component we conducted a simulation study to evaluate different strategies for technical debt management as proposed in the literature. The results of this study provide compelling evidence for current technical debt management strategies proposed in the literature that can be immediately applied by practitioners. Finally, we describe the initial work on an extended simulation framework which will form the basis of a complete *simulation* component for a technical debt management and release planning decision support framework.

INTRODUCTION

Most software engineers understand the concept of technical debt intuitively, even if they are unfamiliar with the formal definition and underlying metaphor. The notion of technical debt can be summarized as the act of making compromises in one dimension (e.g. maintainability) in order to receive a short-term benefit in another (e.g. delivering a release on time). For example, a developer takes a short cut, introducing a code smell [1], in order to meet a release deadline. However, this shortcut compromises the system by incurring a debt that must be repaid to restore the health of the system and to avoid interest in the form of decreasing maintainability.

Motivation

As debt accumulates, it becomes vital to find a way to manage the overall debt such that the system remains both flexible and extensible. This leads to the following questions [23]: When should a debt be refactored? Which debts are easy to fix and promise high gain in software quality? Which debts are hard to fix and promise low gain in software quality? [12]

These questions require a means of decision support and planning. A form of planning, called release planning already exists in software engineering. Unfortunately, release planning is currently limited to the consideration of new feature development or maintenance tasks. This thesis discusses the need for release planning models to incorporate the notion of technical debt as an objective measure to help guide decisions faced by software engineers when prioritizing tasks for the next release. The proposed

approach is demonstrated as a framework that can model different types of changes (new features and maintenance tasks) to understand the types of decisions that can lead to better systems (i.e., with less technical debt).

Summary of the Approach

A multilevel solution to address these questions is proposed. It utilizes a hedonic coalition formation game [7] to distribute work items (i.e., new features, maintenance, or refactoring) to teams and a weighted voting game approach to assign associated tasks to software engineers during the current development cycle. We also use discrete-event simulation to evaluate team technical debt management strategies. This approach comprises the simulation of several teams evolving a project in the presence of technical debt. The results of the simulations corroborate current thought [12][21][24] in the software engineering community with regards to techniques to manage technical debt. Finally we use a combination of the simulation approach with the coalition formation games in an extended simulation framework as the beginnings of a decision support framework which combines the elements of release planning and technical debt management through a common meta-model.

Summary of Contributions

- Unification and extension of existing release planning and technical debt management models to support practical management of technical debt during the entire software development lifecycle.

- A coalition formation algorithmic approach to simultaneously handle both release planning and technical debt management during iterative development planning.
- Development of a Discrete-Event Simulation model used to evaluate and predict the performance of technical debt management methods.
- Empirical results, based on simulation, which confirm current thought in the management of technical debt.

Organization

The rest of this thesis is organized as follows: Chapter 2 provides background on the current issues surrounding the notion of technical debt and its management. Chapter 3 defines the problems at hand for a unified approach to release planning and technical debt management. Chapter 3 also describes a decision support framework which can employ this combined approach. Chapter 4 formally defines the unified technical debt management and release planning meta-model underlying this approach. Chapter 5 presents a multi-level solution to the problems proposed in Chapter 3 using coalition formation games, and explores the effect of this approach empirically. Chapter 6 presents a simulation study concerning practical approaches to technical debt management. Chapter 7 presents current work which extends the simulation approach of Chapter 6 with the conceptual model of Chapter 4. Finally, Chapter 8 summarizes the approach and the experimental results, and provides guidelines for future directions.

4

RELATED WORK

Introduction

This chapter describes the underlying work that forms the foundation for this thesis and the research gaps, which it addresses. Specifically, there are three main areas of contribution: Technical Debt Management (Technical Debt), Strategic Release Planning (Release Planning), and Software Process Simulation Modeling. The current research in each of these areas will be discussed below in addition to any background necessary to further understand this thesis. The contributions and differences between current research and this thesis will also be highlighted.

Technical Debt

The term *Technical Debt* was originally coined by Cunningham [2] as a way of explaining the need to refactor [1] software. The concept is based in a financial metaphor. In essence, taking a short cut in design or implementation, at the expense of a long-term goal such as quality, in order to satisfy a short-term goal like time-to-market is similar to taking out a debt against your software. Interest is accrued (possibly) as long as this debt stays in the code, and the method for repayment is through refactoring the software [1] [3] [4].

Refactoring, according to Fowler et al. [1], is "the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure." They also indicate that there exists "certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring," and that it is

these structures that they called code smells. It was these code smells (and other poor design choices) that Fowler et al. [1], Kerievsky [3], and others [5] [6] [7] [8] identified, that can be attributed as forming a seminal technical debt space.

Research in the technical debt landscape has been focused on the following key areas: 1) definitions of technical debt, the metaphor, and its properties; 2) approaches and techniques to manage technical debt; 3) impact and consequences of technical debt; and 4) methods of measurement. Each of these areas is explored below and its connection to this thesis are explained.

Technical Debt – Metaphor, Definition, and Properties

The notions surrounding technical debt until recently have been informal and under-specified. Recently, Tom, Aurum and Vidgen [9] conducted a systematic literature review in order to consolidate the concepts surrounding technical debt into a single hierarchy. This hierarchy classifies technical debt from either of two perspectives: by the underlying intention behind the decision (or lack thereof) to take on the debt, or by the type of artifact in which the debt occurs.

The intentional perspective is divided into *Strategic Debt, Tactical Debt, Incremental Debt,* and *Inadvertent Debt*. Strategic Debt is debt taken on intentionally as part of a larger long-term strategy. Tactical Debt is debt taken on intentionally as a reactionary response and serves to satisfy short term needs. Incremental Debt is debt taken as several small steps but which accrues very easily and rapidly. Finally, Inadvertent Debt is debt taken on unintentionally and possibly unknowingly by the software development team. The location or artifact perspective is divided into *Code*

Debt, Design and Architectural Debt, Environmental Debt, Knowledge Distribution and Documentation Debt, and *Testing Debt*.

Beyond classifying and understanding of how debt occurs, some researchers have furthered the understanding of the metaphor itself. Nugroho, Visser, and Kuipers [10] indicate that the technical debt metaphor has several contexts from which it can be viewed, and they specifically look at it from the context of maintainability. Along similar lines Klinger et. al. [11] look at technical debt from the perspective of enterprise development and indicate that using financial tools, decision theory, stake-holder based quantification, and developing an understanding of unintentional debt are potential avenues of interest. Finally, Theodoropoulas, Hoffberg and Kern [12] view technical debt from the stakeholder perspective and provide a new definition based on the gap between technology infrastructure of an organization and its impact on quality.

More recent work has looked into the extent and practicality of the technical debt metaphor itself. Specifically, Schmid [13] [14] notes that as we explore technical debt the metaphor begins to breakdown. He notes, the intimate connection between future development and technical debt leads to an inability to objectively measure technical debt itself. This is due to the nature of the interest property associated with technical debt items. Since technical debt interest has a probability which indicates whether it may affect the system, we should instead focus not on measuring all technical debt (potential technical debt) but rather we should concern ourselves with the debt items that will have an impact (effective technical debt) on upcoming feature development or maintenance to be completed.

Finally, current research has focused on how the software industry and practitioners view technical debt. A recent study by Spinola et al. [15] did not find strong agreement among practitioners regarding several notions of "technical debt folklore." Specifically they found that developers unanimously agree that "not all technical debt is intentional." A recent study of industrial practitioners conducted by Codabux and Williams [16] found that there was a lack of consensus concerning technical debt terminology. They also noted that in practice, technical debt decisions are rarely quantified, and that an understanding of the risks concerning long and short term debt requires further investigation.

Technical Debt Management

Technical Debt Management comprises the actions of identifying the debt and making decisions about which debt should be repaid and when. The current industry focus has been on identifying and tracking debt as part of the working project backlog [17] [18] [5] or as part of a separate technical debt list [19] [8] [20]. Essentially, we can think of the emergence of code smells within a code base akin to taking on debt, and the longer they are allowed to remain (without refactoring) the more negative influence they will have on the code base [21]. This influence resonates through the code and makes the software harder to extend and maintain in the future, thus causing developers to pay interest on the debt by increasing the amount of effort required to affect a change [7].

An approach towards developing a technical debt management framework (TDMF), see Figure 1, was put forth by Guo and Seaman [19] [8] [20]. The central concept to this framework is the Technical Debt List (TDL). The TDL stores pertaining

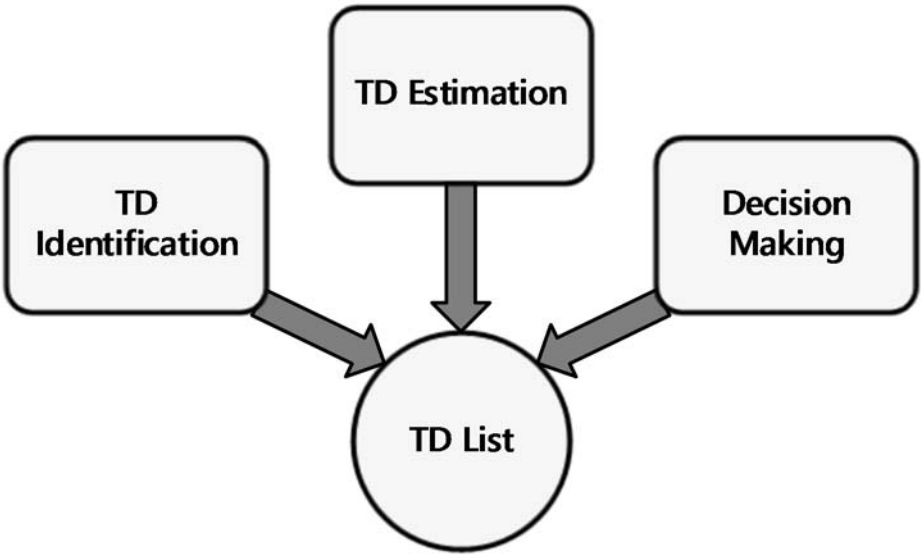

Figure 1. The Technical Debt Management Framework as proposed by Seaman and Guo [17].

to known technical debt items in a software system. Three activities support the TDL. The first is Technical Debt Identification. Here, several tools can be used to find technical debt items which are then automatically placed in the TDL. Each of these items is assigned an informal description stating what it is and where it can be found in the system. The next activity, technical debt estimation, assigns to each item estimates for the debt principal (cost to remove) and the interest (additional cost due to potential future problems they may cause alongside a probability that this event may occur). Each estimate is provided a Likert scale value of low, medium, or high. Other information such as correlations to other debt, the interest standard deviation, and the date it was inserted into the list can also be assigned. The final activity, decision making, is used to help assess which debts should be addressed (via remediation) and when they should be addressed.

The need to manage and understand the decision process concerned with technical debt is currently at the forefront of research. This research is currently divided into two sections: decisions concerning repayment of debt and decisions concerning when to take on a debt. For the former, Zazworka, Seaman and Shull [22] have attempted to determine the decision processes that indicate when different debt should be paid off according to a prioritization based on cost of the debt and its impact on quality. Further, Codabux and Williams [16] note that a practical and effective technical debt strategy is to have dedicated teams with the purpose to reduce technical debt, while product development teams devote 20% of their effort towards debt reduction.

Recent work by Schmid [14] [13] [23] has focused on a cost estimation based approach towards selecting which debts should be removed. Schmid's work is based on a formalization of technical debt concepts to extend the TDMF. This framework utilizes a 2D matrix representation and approximation scheme to identify which technical debt items to include in the next release. In a similar vein Stochel, Wawrowski and Rabiej approach the problem using a subsumption model of technical debt [24]. This model views technical debt from three perspectives Code/Design debt, Architectural Debt, and Portfolio Debt wherein each of these levels aggregates those contained within. This model utilizes a modified Value Based Software Engineering [25] cost and estimation approach in order to estimate the ROI for each of the technical debt items. The goal is then to utilize a technical debt versus portfolio assessment matrix, using ROI in a similar approach to that of Seaman and Guo [8], to determine which technical debt items provided the best savings per release (similar to that of Schmid [14] [13] [23]).

In a further enhancement to the TDMF, Holvitie and Leppänen have developed an approach known as DebtFlag [26]. The purpose of the DebtFlag is to automate the association of technical debt items in the TDL with their incarnations in the code. They also utilize automated debt propagation to better enhance the estimation of debt impact, interest, and interest probability.

The approaches to decision support for selecting when to take on additional debt have been less forthcoming than those for repaying debt. Falessi et al. [27] are exploring current open problems surrounding this topic as well as the required decision support constructs needed to address the problem. Ramasabba and Kemerer [28] began work towards an optimization approach utilizing multiple projections of a single codebase in order to help software engineers make decisions regarding both acquiring and repaying debt. Eisenberg [29] also argues the need to utilize thresholds as a means to warn developers and managers that their technical debt may be getting out of hand. Thus, there is a need to explore which thresholds are needed. Eisenberg's approach can also be utilized to help determine if the product can withstand taking on additional debt as well as help determine when debt must be repaid.

Impact and Consequences of Technical Debt

The impact of technical debt on engineering effort, project cost, and project quality is of utmost concern as well. Zazworka et. al. [7] conducted an initial empirical study into the effects of technical debt on quality. They showed that the form of technical debt they studied (God Classes) has a negative impact on software quality. Further, Zazworka, Seaman, and Shull [22] attempted to extract the decision process that can be

used to prioritize which debts (based on cost/benefit analysis) should be paid off. This study showed that technical debt negatively affects the correctness and maintainability of a product (using defect likelihood for correctness and change likelihood for maintainability).

A key to understanding technical debt and its effects is to be able to understand the gaps and overlaps that may exist in the landscape [30] where various approaches are exercised. Zazworka et al. [6] identify several types of design debt (e.g., code smells, modularity violations, and design pattern grime) and tools which detect them. They identified that all the tools indicate different problems with little to no overlap. Fontana, Ferme, and Spinelli [21] state that although code smells are important components of the technical debt landscape, certain identified debt items may not actually constitute debt. Instead they indicate that domain knowledge must be used as a filter in order to identify these misnomers and to ensure that an accurate indication of technical debt is provided.

Morgenthaler et al. [31] conducted a study at Google and identified a new form of debt called Build Debt. Build Debt involves inefficiencies and impact on the quality of the build process. This study evaluated not only the impact to the project, but also the impact to the developers, while providing some insight into the management of such debt.

Technical Debt Measurement

Lastly, there must be a means to measure technical debt and its associated properties in a way that is both meaningful to developers and to stakeholders alike. Seminal work by, Brown et. al. [18] identified the technical debt metrics of: principal,

interest probability, and interest amount. Subsequently, Nugroho, Visser, and Kuipers [10] contributed a formal model to calculate measurements for both interest and principal. Additional measures, closely related to the technical debt landscape [5] [6] have been proposed to index the effect that design flaws (e.g., code smells and modularity violations) have on technical debt. For example, Marinescu [32] proposes a method to index the effect on quality produced by different code smells and anti-patterns based on the type, influence and severity of the design flaw instance, thus creating a score which can be aggregated over the size of the system. In another approach Nord et al. [33] develop a strong foundation for measuring the architectural technical debt based on the notion of prudent, deliberate, and intentional debt.

Letouzey [34] developed the SQALE quality and technical debt analysis model which not only provides the ability to estimate technical debt principal but also provides several visualizations to help developers and management understand and analyze the impact of technical debt in their projects. Recently, Curtis, Sappidi and Szynkarski proposed methods to estimate the principal and interest [35] as well as the size, cost, and type [36] of technical debt. Given these various approaches for the quantification of technical debt and the wide range of differences in values, Izurieta et al. [37] proposed a means to measure the error associated with the calculation of technical debt for these methods. They argue that a means to measure the systematic error introduced by these tools should be included with their values, similar to other scientific tools, and that a means to compare these tools and their error be developed.

Release Planning

Release planning is divided into two problems, which correspond to the two levels of release planning. The first is the strategic release planning problem, which deals with the partitioning of a set of work items (e.g., new features or user stories) into a set of releases such that a set of constraints are satisfied [38]. The second is the operational release planning problem, which deals with the allocation of tasks to software engineers under a set of time and capacity constraints. Several works by Ahmed Al-Emran et al. [39] [40] [41] [42] [43] [44], Saliu and Ruhe [45], Huang and Chiang [46], and others have looked into approaches to addressing the problems surrounding this area. Typically these approaches involve the computational intelligence approaches such as genetic algorithms [47] [48] [49] [50] [45] and ant colony optimization [46] [51]. Al-Emran et. al. [40] [44] [40] [43] have combined their approach with a discrete-event simulation model, called DynaRep, in order to combine computer intelligence with human intelligence to aid in the decision making process. This field has developed in several areas [52] including: *feature elicitation*, *problem specification*, *resource estimation*, *stakeholder voting*, *release plan generation*, and *evaluation of plan alternatives*. The research described in this thesis is concerned with release plan generation in conjunction with technical debt management.

Strategic Release Planning

Strategic release planning encompasses the “optimum” selection of feature or requirements under a set of constraints to be delivered during a given release [40] [53]. In essence, strategic release planning can be seen as the partitioning of a product backlog

into several sprint backlogs in a Scrum [54] setting. This problem stems from the work of Bagnall [55] in optimizing for the selection of requirements to be completed in the next release. Additional work has concentrated on two approaches: planning for the next immediate release, or planning for the next *n* releases.

This problem, which has been shown to be NP-Complete by reduction from the Knapsack Problem [56], has been approached in two distinct ways. The first approach is through Integer Linear Programming (ILP) [56] [57] [55] [58]. It has been shown that the ILP based approaches have an issue with scaling when the number of requirements becomes large [59]. Due to the lack of scalability inherent in ILP approaches, alternative approaches using computational intelligence have become prominent [60] [49] [55] [61] [62].

Because of its computational difficulty, there is evidence to suggest that this problem falls into the category of wicked problems [63]. Wicked problems are problems where regardless of the ability to optimize a solution it may still not be a very good solution when it comes to actual implementation of the solution. This has led to a trend of hybrid systems combining both computational and human intelligence [50]. The goal of this research is to present the user with a selection of "good" release plans and let them decide which will be the best to implement.

It should be noted that with all of the approaches developed to date, relatively little industry adoption has taken place [64]. This is further explored by Svahnberg et al. [64]. They indicate that the issue is not the existing models/approaches, but rather the limited amount of industry grade models/approaches.

Operational Release Planning

Operation release planning is the problem of allocating resources to tasks such that constraints on developer availability, task dependency, and budget constraints are met such that the overall time to release is minimized. A special case of this problem considers the software engineer resources. This problem is known as the project staffing problem [59]. Ruhe [59] has identified three distinct sub-problems, in the project staffing problem, each with different end goals:

- *Planning to maximize the total release value*. Here the goal is to maximize the number of work items completed such that the value of the system (as evaluated by external quality indicators (such as stakeholder satisfaction)) is within a fixed time interval. It should be noted that this problem is a combination of both the *knapsack* and *job-shop scheduling* problems [59], each of which is known to be NP-Complete [59].
- *Planning to minimize release make-span.* Here the goal is to minimize the amount of time required to complete a set of work items assigned to a given release.
- *Planning for maximum competency match*. Here the goal is to maximize the matching between task and developer such that the developer's ability (as measure by their productivity or knowledge level) to the task, in order to ensure that the overall release time is minimized. This problem has been shown to be a special case of the *resource-constraint project scheduling* problem, which is known to be NP-Complete [59].

In each of these problems the main components are the set of tasks (derived from the work items assigned to a release) to be developed, the software engineers working as a part of the team assigned to the release, and finally the set of productivities for each individual developer for each type of task [59].

Operational release planning falls into the area of project scheduling which is a subfield of project management. Each of these areas have been well studied, see Blazewicz et al. [65] for an introduction to project scheduling and Wysochi [66] for an introduction to project management. Early work in the area of optimal project scheduling and operational release planning was conducted by Chang et al. [67]. Chang et al. studied the problem of project scheduling using an approach based on genetic algorithms. This approach focused on the problem of schedule minimization but did not take into account developer productivity for given types of tasks.

Another early approach was that of Abdel-Hamid [68]. Abdel-Hamid suggested the use of a system-dynamics simulation model of the development process to produce project schedules. Similarly an approach by Fenton et al. [69] proposed the use of Bayesian belief networks as a means to manage the inherent uncertainty in effort estimation techniques use during the decision making process.

More recently, there has been a return to search-based techniques when dealing with the project scheduling problem. Specifically, Antoniol et al. [70] evaluated by comparison a range of techniques including hill-climbing, genetic algorithms, and simulated annealing. They found that of the techniques applied, simulated annealing was the best and that this approach reduced the project make-span by 43% when compared to

a baseline of random search. Another approach, by Barreto et al. [71], modeled the problem as a constraint satisfaction problem. Although they took into consideration characteristics of the project, including developer competency, they failed to explore the scalability of the approach, something that is noted by Ruhe [59] as an issue of several approaches.

Ruhe [59] suggests that the overall complexity of this problem requires a solution which is more formal and scalable than what has been already put forth in the literature. Given this, Ngo-the and Ruhe [57] have developed a two phase solution which can be used to solve both the *planning to maximize the total release value* and *planning to minimize release make-span* problems. In the first phase, they use an integer linear programming approach to solve a reduced form of the problem. Using this solution, they apply a genetic algorithm to find the best assignment of developers to tasks. This full approach is necessary to solve the first problem, but in order to handle the second problem only the second phase is required (which they suggest using the genetic algorithm proposed by Hartmann [72]). As for dealing with the problem of *planning for maximum competency match*, they note that a greedy approach, as proposed by Rahman et al. [73], is the best approach.

Recently, Przepiora, Karimpour, and Ruhe [74] connected earlier efforts based on genetic algorithms with constraint programming in order to evaluate the effectiveness of constraint programming on the release planning problem. This approach evaluated a pure constraint programming approach against the combined approach. They found that in less complex situations the use of constraint programming is not necessarily the best solution,

but in the situations where it is, the combined method was 87% more efficient than the constraint programming approach alone. These comparisons were conducted on a data set generated using industry data. Another approach by Nayebi and Ruhe [75] has connected the release planning and prioritization methods of prior work to an open innovation (crowd-source) approach and provide a proof-of-concept evaluation.

Along with these approaches to solving the operational release planning problems, there has been research into the effects of uncertainty in estimation on the operational release planning process. Specifically, Al-Emran et al. have studied the effect of changes in the number of features, the number of developers, effort estimations, and productivity estimations on operational release plans [39] [42] [43]. Initial work was conducted across simulated data and showed that a 20% variability in effort estimation can have a profound effect on the execution times of tasks and can require more than 50% developer reassignment [39]. Later studies showed that in the worst case (50% effort over-estimation and 30% developer dropout) increased the release make-span by a maximum of 50% [42]. Using data from industry they showed that the effect of a combination of factors was always greater than the summation of the individual effects.

Release Re-Planning

In 2000, van Lamsweerde [76] completed a survey of over 8000 software development projects from 350 US software development firms. The results of this study showed that of the projects studied one-third were never completed and only one-half partially succeeded. Van Lamsweerde also showed that a major source of failure (11%) was changing requirements. Kotanya and Sommerville [77] found that for market-driven

software the following are the major causes of frequent changes in features and requirements: errors, conflicts and inconsistency in requirements; changes in customer or end-user knowledge about the system; technical, schedule, or cost issues; changes in customer priorities; environmental changes; and organizational changes.

These issues have given rise to the need for release re-planning. Release re-planning can be defined as the process of modifying an existing release plan in order to accommodate unforeseen changes [59]. Initially Albourae, Ruhe and Moussani [78] introduced the concept of release re-planning using a greedy approach. They also identified the release re-planning problem as a form of knapsack problem [79]. Their greedy approach was focused on the use of the analytical hierarchy process [80] using rough estimates of effort and stakeholder voting.

Further work by Al-Emran, Pfahl and Ruhe [44] combined operational release planning with release re-planning through discrete-event simulation and called this approach DynaReP. Using this model they could evaluate the effects on changes in the number of features, number of developers, effort estimates, productivity estimates, and execution time estimates. Combining this method with risk analysis capabilities, Al-Emran and Pfahl [40] where able to evaluate the effects of different worst case scenarios. This later effort was limited to simple examples and did not compare itself to any other approaches. Al-Emran, Pfahl, and Ruhe [81] combined DynaReP with another operational release planning process to compare plans against an initial baseline. This provided a form of sensitivity analysis for operational release plans in the face of the types of change that would prompt for re-planning.

Jadallah et al. [82] developed an underlying process to release re-planning which focuses on answering How? What? and When to re-plan? (H2W). Using these questions as a basis they devised, and showed proof-of-concept of, a greedy approach for release re-planning. Al-Emran et al. [41] extended the H2W method by combining re-estimation and re-planning together, calling the revised approach H2W-Pred. This new approach incorporated dynamic updates of defect and effort estimates during the re-planning process. They showed that including re-estimation can yield a portfolio of solutions, which can balance and compare trade-offs between functionality and quality for several release plans. Ruhe [59] describes a refinement to H2W, called H2W*, which enhances the underlying model, and provides empirical results validating the method.

Finally, Golforelli, Rizzi and Turricchia [83] have looked at the combination of operational release planning and release re-planning from the context of the agile development process Scrum. Their approach uses a linear programming model for the operational release planning problem and a minimum perturbation strategy for the release re-planning problem. They evaluated their method across a set of 58 synthetic projects evaluated using the model implemented using a commercial linear programming system. Although motivated by the need for robust tools for agile development, their model is lacking support for multiple teams, developer productivity, developer skill, and evaluation of execution time. It should be noted that this approach considers soft-constraints (e.g., ensuring that the tenants of Scrum are met), which Svahnberg et al. [64] notes as missing in most release planning models evaluated. They also note that this

approach can handle only small to medium projects (those with approximately 100 user stories).

Game Theory

Game theory can be seen as a dynamic form of decision theory. Here we have a group of agents. The goal for each agent is to maximize their utility under the constraints of the game at hand.

Game Theory and Software Engineering

Game theory has been used to model various aspects of software engineering. For instance, Bacon et al. [84] used a non-cooperative game model between a worker and a manager to explore the use of incentives in software development. Other approaches have made use of *mechanism design* to improve the software engineering process through the use of incentives [85] [86]. Another example can be found in the area of automated refactoring, where Bavorta et al. [87] use a non-cooperative game to identify *Extract Class* [1] refactoring opportunities.

Coalition Formation Games

This research focuses on coalition structure formation wherein coalitions are formed based on maximizing a preference function defining the payoffs of each player for the available coalitions [88]. A coalition formation game, G, is defined as $G = (\mathcal{N}, \prec)$. Where $\mathcal{N}$ is the set of players or agents and $\prec$ is the preference relations over the coalitions for each player [88]. The types of coalition formation games we are

specifically concerned with are hedonic coalition structure formation games, or hedonic games.

Hedonic Games. In Hedonic Games the formation of coalitions is constrained to attain Pareto Optimality [88]. The condition of Pareto Optimality states that no change may be made to increase a player's payoff without reducing another player's payoff. Bogomolnaia [89] simplified the concept of hedonic games such that a player's payoff is solely based upon the other players within the coalition. In this work we extend the algorithm proposed by Saad et al. [90] [91] in order to produce a set of possible solutions (rather than only one) and to be able to include constraints (which may violate the Pareto Optimality condition) in order to allow forced positioning of players within coalitions.

Weighted Voting Games. Weighted voting games form coalitions by allowing each player to vote for a specific strategy, where each player's vote has an associated weight [19]. The players can then form coalitions using their votes. The coalition whose number of votes exceeds a predefined threshold is selected as the winner [19].

Software Process Simulation Modeling

Software process simulation modeling (SPSM) is a branch of empirical software engineering, which focuses on simulating different aspects of the software development life cycle. Simulation has been widely used as a means of prediction and analysis in the software industry [92] [93]. Kellner, Madachy and Raffo [93] explored the area of SPSM in order to understand the methods used and the problems to which simulation has been applied and to connect the use of simulation to empirical study. They identified that

simulation can be used for, or help facilitate, the following processes: *strategic management*, *planning*, *control and operational management*, *process improvement and technology adoption*, *understanding*, and *training and learning*. In conducting a survey of the literature, they found that most simulation studies conducted are centered on the process or project level.

A further study by Zhang, Kitchenham, and Pfahl [92] on the current trends in SPSM and noted that of all the simulation paradigms used, both *discrete-event* and *continuous* simulation formed the mainstream SPSM approaches (as opposed to *agent-based*, *mathematical*, or *monte-carlo* simulation methods). They also note there is a need to increase modeling and simulation at the process and entity level.

A specific instance of process level simulation is exemplified in the work of Magennis [94], which uses Monte-Carlo simulation to evaluate the effects of changes on agile development processes. Another example of agile process simulation is the work of Glaiel, Moulton, and Madnick [95] which used a System Dynamics model (a form of continuous SPSM) to describe and evaluate agile processes. An instance of entity level simulation is exemplified in the work of Spasic and Onggo [96], which uses agent-based simulation to evaluate the software development processes at AVL.

Ruhe [97] identifies SPSM as a fundamental component of any software engineering decision support (SEDS) system. Because of this it has been widely used to support operational release planning as well as release re-planning [39] [40] [41] [42] [81] [43] [81] [44] [82] [59] [98]. It has been used as both an underlying approach to operational release planning [40] [43] to evaluate how plans can change in response to

uncertainty in the planning process [39] [40] [41] [42] [43]. Specifically, Al-Emran et al. [39] [40] [43] studied the effects of changes in the number of developers, number of features, effort estimates, and productivity estimates on release *make-span* using monte-carlo simulation. They confirmed that such changes can have profound impact on the overall time required for a release.

Contributions

Given the body of work in technical debt management and the academic community's desire to provide solutions which can be put to practical use in the field, there is still a significant lack of empirical work validating the proposed approaches. The difficulty in validating these approaches in practice is due to the difficulties of mapping experimental conditions to real world scenarios for each approach. Yet, as Falessi et al. [27] indicate, the use of simulation and the ability to pose "what if" questions can shed light on such issues as time-to-market or technical debt impact on a system, is a necessary component of technical debt management.

The problem of validating approaches also seems to fall into the problem space that Kellner, Madachy, and Raffo [93] indicate as an issue apt for SPSM. Given this we have developed a simulation model for validating practical approaches for technical debt management. The framework allows for parameterization of sub-components, such as the incorporation of the TDMF as well as looking at whether it is better to use a dedicated sprint for TD removal or a percentage per sprint. A customizable approach is critical in order to gain insights about which strategies are better for a given software development organization. The use of simulation for technical debt management has yet to be

explored, and the hope is that the seminal work described herein provides strong motivation to continue using simulation.

Along with the problem of validating proposed techniques, advanced technical debt management approaches (such as those presented by Schmid [14] [23] and Stochel, Wawrowski, and Rabiej [24]) fail to take into account the surrounding development process in which these practices are introduced. It should also be noted that despite extensive literature reviews, there does not appear to be an approach which combines the use of advanced release planning techniques with technical debt management. Given that the ideas surrounding the need for tools supporting technical debt management are in line with those used in release planning (such as what-if analysis and other decision support concepts) [27], and the fact that TDMF has a decision support component, a combination of release planning techniques and technical debt management is a natural progression of the current work. Thus, this thesis puts forth a unified model which connects the concepts of TDMF and release planning.

There exists little empirical work validating the practical methods of technical debt management proposed by researchers and industry practitioners. Along with this is the lack of empirical validation of the more advanced techniques meant to provide efficient and accurate methods of technical debt management. The first problem is that we assume that these more advanced methods are superior and efficient than those methods found in industry today, yet there is no work currently validating this assumption. To date, there is a dearth of research that evaluates these different methods

against one another. To compare and contrast advanced methods we provide a framework, but leave the analysis to future work.

DECISION SUPPORT FRAMEWORK

Introduction

This chapter defines a number of problems which are important to both release planning and technical debt management. Once identified we can see how the problems in these two distinct areas are similar or even the same. Given these similarities we then merge the problems into a set of combined problems, which can then be mapped onto the meta-model defined in Chapter 4.

Problem Definitions

Technical Debt Management

There are two main problems stemming from the study of technical debt management. The first is determining at which point a project should repay a debt through refactoring. The second problem is in determining at what point a project should take on new debt. Both of the problems described in the following subsections apply only to known and tracked technical debt items or the creation of intentional debt.

Technical Debt Management – Remediation (TDM-R). The question at hand is: When should a debt be repaid? The problem is that even though technical debt must be dealt with, new feature development and maintenance of existing code must be conducted as well. Thus, the selection of the highest value (those which affect the largest amount of existing code and new feature items) technical debt items with the lowest cost (in effort) should be selected [22]. This is essentially an optimization problem where we are attempting to maximize the amount of technical debt removed (focusing on tactical,

incremental, and inadvertent debt*), minimize the cost of this removal*. This problem is fairly straight-forward when considered alone, but unfortunately it must be considered within the larger context of software development lifecycle.

Technical Debt Management – Debt Acquisition (TDM-DA). This problem deals with the acquisition of intentional debt. From the discussion in Chapter 2, there are three forms of intentional debt: strategic, tactical, and incremental. The overarching question here is: *When should new debts be acquired?* Strategic debt acquisition involves a decision point during strategic release planning, which if accepted will allow the increase in the number of new features to be developed during the next *k* releases but at the cost of long-term software issues. In the case of tactical debt, this seems to be the problem of identifying the point within a release that has the following two outcomes: i) allow the release constraints (cost or make-span) to be violated, or ii) take on a new debt to ensure that the constraints are not violated. Incremental debt acquisition involves the decision during development to limit refactoring in order to continue new feature development. Strategies for handling incremental debt, such as devoting a percentage of software engineer time during a release or using thresholds to control the amount of debt accrued [17], are explored in the simulation study found in Chapter 5.

Strategic Release Planning (SRP)

Strategic release planning, as described in Chapter 2, is the "optimum" selection of features or requirements under a set of constraints to be delivered during a given release [40] [53]. The question underlying this problem is: *Which partitioning of a set of work items into k releases will satisfy a given set of constraints?* Where the constraints

are *dependencies between work items*, *minimum cost* or *minimum make-span for each release*, and *maximum number of high priority work items completed*.

Operational Release Planning (ORP)

As discussed in Chapter 2, operation release planning is the problem of allocating resources to tasks such that a set of constraints are satisfied. We are mainly concerned with a special case of this problem: the project staffing problem. Further, we discussed a set of three sub-problems of the project staffing problem: *planning to maximize the total release value* (ORP1), *planning to minimize release make-span* (ORP2), and *planning for maximum competency match* (ORP3).

Release Re-Planning (RRP)

As discussed in Chapter 2, release re-planning is the problem of determining the when, the how, and the what of re-planning an existing release plan. These questions are well developed by Jadallah et al. [82]. For this thesis we will consider these questions as simply the release re-planning problem (RRP).

Combining Technical Debt Management and Release Planning

These problems are very similar at an abstract level. The differences between the problems lie only in the constraints and the point in the development lifecycle at which they become pertinent. Table 1 shows the mapping between problems and their associated constraints and Figure 2 depicts the connection between software development lifecycle timing and the different problems.

Table 1. Mapping of constraints to the RP and TDM problems.

	Constraints						
Problem	*Max. TD Removed*	*Min. Release Cost*	*Max. Release Quality*	*Max. Release Value*	*Max. Work Satisfaction*	*Min. Make-span*	*Max. Competency*
SRP		X	X			X	
ORP1		X	X	X	X		
ORP2		X	X		X	X	
ORP3		X	X		X		X
TDM-R	X						
TDM-SRP	X	X	X			X	
TDM-ORP	X	X	X	X	X	X	X

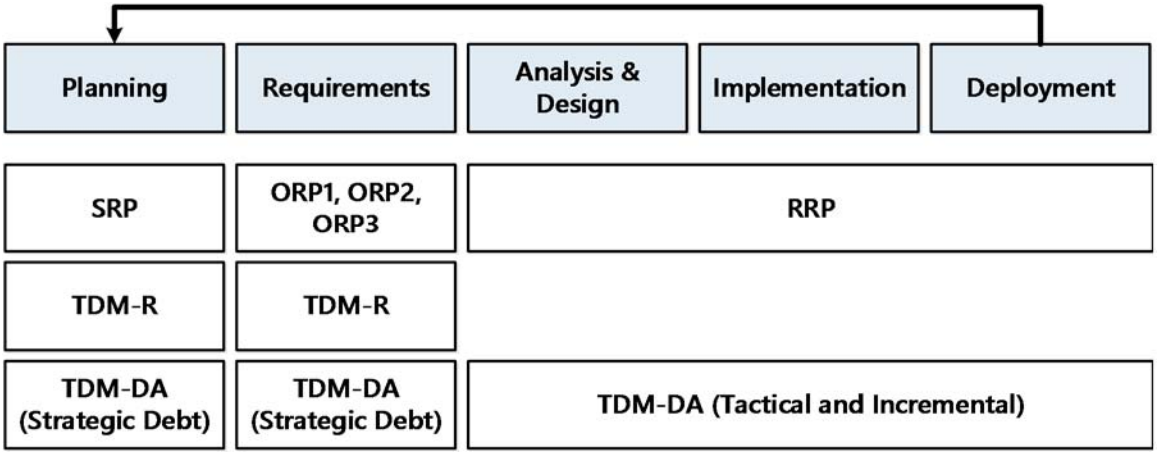

Figure 2. Mapping of problems to phases of the generic iterative software development life cycle.

We first begin with the SRP, the TDM-R and the TDM-DA problems, since each of the problems (or a portion of each problem) look at the long term future of the software product. In the case of SRP we are looking at the most desirable features to implement before a series of releases are completed. In the case of TDM-R we are

considering which technical debt items to remove before a series of releases are completed. Finally, in TDM-DA we are looking to determine if taking on strategic debt will be beneficial during the next few releases.

We can then reduce the TDM-R and SRP problems into a single problem: *Technical Debt Management in Strategic Release Planning (TDM-SRP)*. Here the problem is to partition the set of technical debt items and new features into a set of k releases. The constraints of this problem become: *maximize technical debt remove, minimize release cost* or *minimize release make-span,* and *maximize release quality*.

At this point we are still in release planning, but the software engineers and management may be considering choices which will create strategic technical debt (such as a choice in development technology or environment which will lead to a technological gap). Though this is at seemingly at odds with the *minimize technical debt* constraint, but this is not so. Intuitively, in the planning phase, we are looking to maximize the amount of technical debt removed (with minimal cost) in the next *k* releases, but this does not preclude the acquisition of new debts, if those debts can help satisfy other constraints such as minimizing release make-span, minimizing release cost, or maximizing release value, since each of these problems are by their very nature multi-objective.

In the TDM-DA problem the entity of concern is a *decision variable* with multiple outcomes (which theoretically we can measure the value of). If we include this type of item we can then partially include the strategic TDM-DA problem within the TDM-SRP problem, and this can be completed by changing the constraints. A solution to the TDM-SRP problem will then solve the TDM-R, strategic TDM-DA and SRP

problems, but it leaves the open the ORP, the RRP, and the tactical and incremental TDM-DA problems.

At the operational level, release planning focuses on a single release and has the goal of matching software engineers (or other resources) to a given set of tasks within a set of constraints. In ORP, the main entities of concern are Tasks and Resources (Software Engineers). The first step is to formalize the three ORP problems as a single multi-objective problem: *Find the optimal matching of software engineers to tasks of a given release such that dependencies between tasks are satisfied, release cost is minimized, release value is maximized, release make-span is minimized, software engineer task preference matching is maximized, and software engineer satisfaction with assignments is maximized*. Given these constraints we include technical debt with the following constraints: *maximize the amount of technical debt removed*, *maximize system quality at completion of the release, minimize release cost* or *maximize release value* or *maximize engineer to task competency matching, maximize engineer work satisfaction,* and *minimize release make-span*. This problem becomes the TDM-ORP problem. At this stage all that is left are the problems surrounding release re-planning and managing decisions concerning tactical and incremental technical debt.

Jadallah et al. [82] identified in their approach the three main questions of release re-planning: How to re-plan? What to re-plan? and When to re-plan? The how refers to the underlying method which can be one of many approaches. The interesting questions for this research are in the “What” and “When” questions. Here we need to merge the TDM-DA and RRP problems. In TDM-DA we are concerned with the decisions to be

made, and as identified previously we can encode the choices and their values as a type of work item. Doing this will handle the decisions such as when to take on tactical or incremental technical debt, since these are intentional. We can then map these into both the "What" and "When" sections of release re-planning. Since release re-planning occurs during the development phase (even if only simulated) it allows us to deal with decisions regarding technical debt acquisition as well as dealing with changes in developers or features. In the case of tactical technical debt we may take these debts as development continues if the ability to satisfy the release constraints is placed in jeopardy. Simultaneously, incremental debt will continue to build up and affect the overall technical debt level of a system which could trigger a re-planning event (the "When").

Finally, as work items are completed there will be inadvertent debt created, which will increase the overall technical debt level as well, possibly triggering a re-planning event. Other issues such as engineer availability changes, stakeholder priority changes, over/under estimation probability changes, engineer productivity changes, large changes in technical debt levels, or large deviations between reality and the simulation can also trigger re-planning. If a re-planning event is triggered those technical debt items marked as effective and those features with high priority should be the first to be moved into the current release, but without affecting the work already completed.

Decision Support Framework

Given the problems described above, we have developed a software engineering decision support (SEDS) framework based on the initial SEDS concepts from Ruhe [97] relating to release planning. This framework is designed to solve each of the above

problems within the context of the software development process and to consolidate the technical debt management needs with those of release planning. As technical debt builds up in a project the need to make decisions which can affect release dates or push back features come into play. Thus, we have developed an architecture that will serve managers and researchers in helping make and understand these important decisions.

Ruhe [97] demonstrated the need for decision support systems in software engineering, while focusing mainly on the issues surrounding the area of release planning. More recently, Falessi et al. [27] indicated that there is a need for similar decision support in the area of technical debt management. In this thesis we show the development of an underlying meta-model which combines both release planning and technical debt management concepts together and how release planning at both the strategic and operational level can be used to help make decisions regarding the management of existing technical debt. Yet, the integration of both strategic and operational release planning with technical debt does not allow for the necessary analysis that will be required by project managers to evaluate scenarios related to events that occur during the development phases beyond planning. In order to handle this, Ruhe [97] indicates the need for a simulation model which can be used to perform these "what-if" type analyses.

In the following chapters we describe the development of a combined meta-model which unites the fields of release planning and technical debt management. Using this meta-model we propose algorithms which can provide a portion of the analysis and decision component (in conjunction with human intelligence), as well as the intelligence

component. We also demonstrate the use of simulation in evaluating technical debt management approaches and further extend this simulation as to the level necessary to be used as the simulation component in a combined TDM and RP SEDS architecture.

Figure 3 is a depiction of the architecture of such a decision support system. Here, the rounded rectangles are components of the system, the rectangles are external inputs, and the parallelograms are questions which can be answered using the system. The rounded rectangles with dashed lines are considered outside the scope of this thesis but will need to be considered to have a complete framework. As shown in the highlighted sections of Figure 3, this thesis is co ncerned with the *simulation*, *intelligence,* and

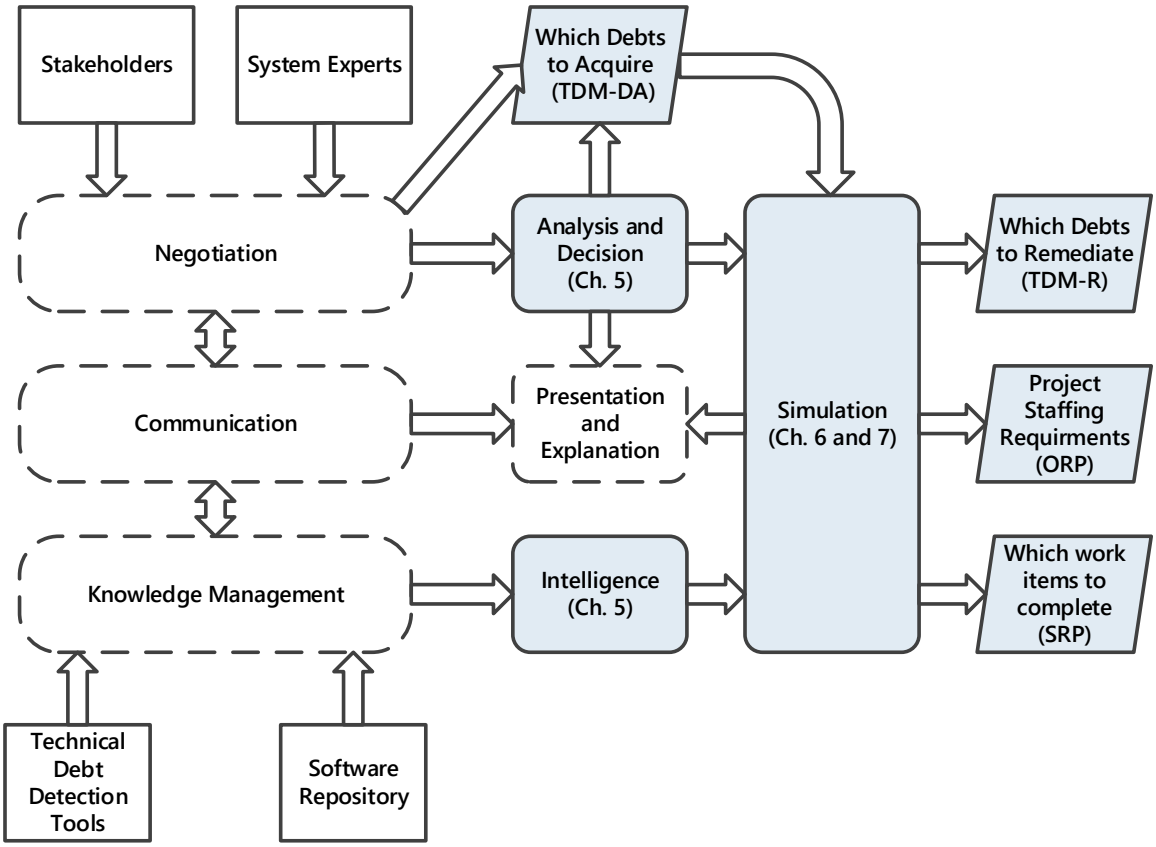

Figure 3. Decision Support System architecture for software engineering decision support in the areas of technical debt management and release planning.

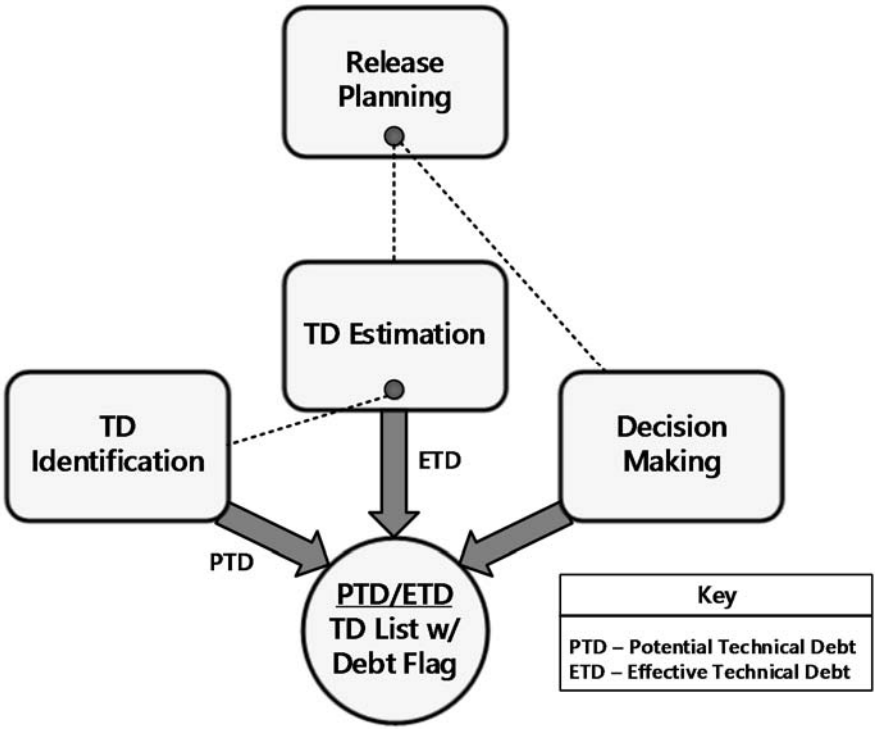

Figure 4. The combination of release planning and the TDMF [17], (where the dashed lines indicate information dependencies between processes).

analysis and decision components. We also show the connection between the questions of concern and the SEDS components.

Finally, Figure 4 shows the extedned TDMF. In this extension the TDMF has been modified to incorporate release planning. It is this extension which forms the basis behind the combined meta-model described in the following chapter as well as the simulation components described in Chapters 6 and 7. The main body of work this thesis provides is the decision making component and the connection between release planning and technical debt management.

UNIFIED META-MODEL

Introduction

This chapter describes a unified and extended conceptual model for the combination of technical debt management and release planning. This unification is based on existing models in release planning [44] [48] [55] [99] [100] [59] [47] [38] [50] [98] [45] and the formalization of technical debt concepts by Schmid [14] [13] [23], which is itself an extension of the TDMF [19] [8].

Unified Meta-Model

The unified model is a meta-model describing the concepts from both release planning and technical debt. There is an underlying difficulty in merging the concepts from these two research areas rooted in the level of information/knowledge required in both areas and in the fact that release planning typically deals with work not yet implemented while technical debt arises only in existing artifacts. Combining these two levels such that both concepts can be characterized yielding a practical model upon which a solution to both problems is possible. We have created such a model, for which the highest level concept is the *System*.

The System

The System (see Figure 5) represents all the components that make up the software. This includes the design, code, software engineers and the stakeholders. In this model, the system is composed of a *repository* containing the design documents and

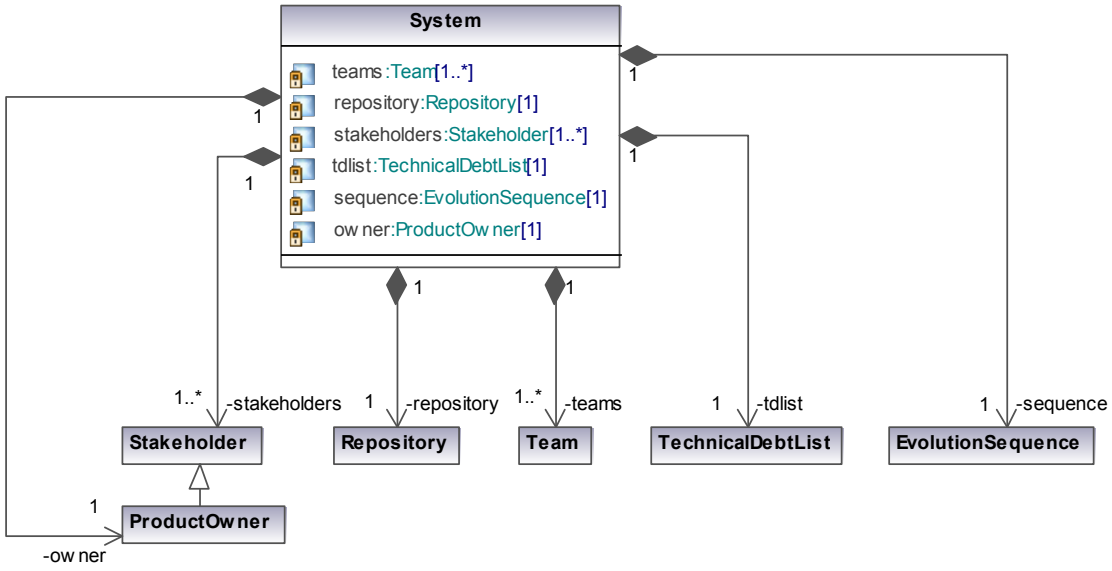

Figure 5. Meta-model section describing the components of a system.

code, an *evolution sequence* (i.e., the backlog), a *technical debt list*, a set of *teams*, a *manager* (i.e., product owner), and a set of *stakeholders*.

Teams

Each team (see Figure 7) is a set of *software engineers* assigned to work on a software system. Teams are described by a set of properties including the *type* of team, a team *productivity* value, a set of *preferences* over work item/task types, a total *potential effort* available, and a *total effort remaining* value.

The team productivity and effort values are derived from the software engineers composing the team. In the case of team productivity, this is the weighted (depending on software engineer type, i.e., Entry-Level, Mid-Level, Senior, etc.) average productivity across all team members. The effort values, on the other hand, are weighted summations across the members of the team. The team type is one of *implementation, testing,*

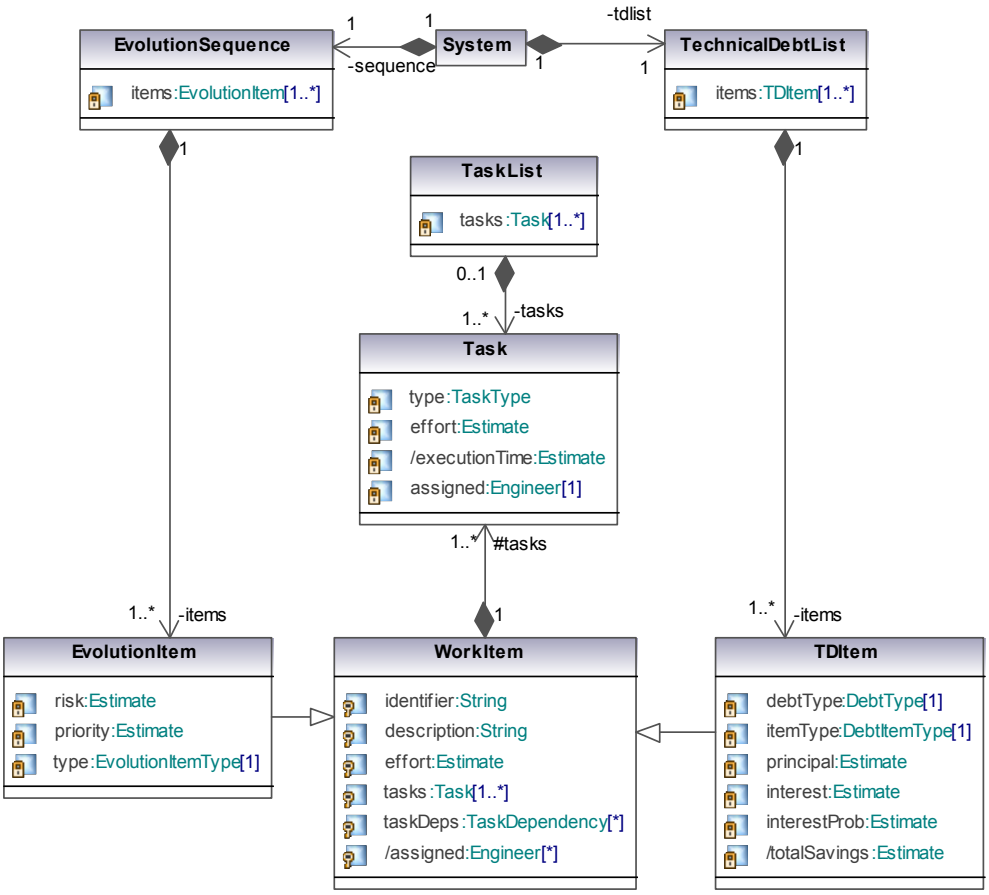

Figure 6. Meta-model section representing the components of the evolution sequence and technical debt list.

technical debt remediation, or *maintenance*. The different team types indicate the preference profile for work item types. Finally, one member of the team is selected to be the team *lead* who serves as the formal point of communication between the team, the manager and other stakeholders.

Evolution Sequence and Technical Debt List

The evolution sequence represents the changes to be applied to the software system. It is essentially a set of *work items*, specifically *evolution items*. The technical debt list is composed of *technical debt items*. It is maintained as a means to estimate the

amount of technical debt currently in the system. Figure 6 describes the components and connections of the evolution sequence and technical debt list within the meta-model.

Work Items

Work items (see Figure 8) are the basic unit of concern in release planning. Each item has a common set of properties: a *unique identifier*, a *description*, an estimate of the *effort* required either to implement the evolution item or refactoring to remove the technical debt item, and a set of *tasks* which compose the required work. In this model there are two specific types of work items: *evolution items* and *technical debt items*.

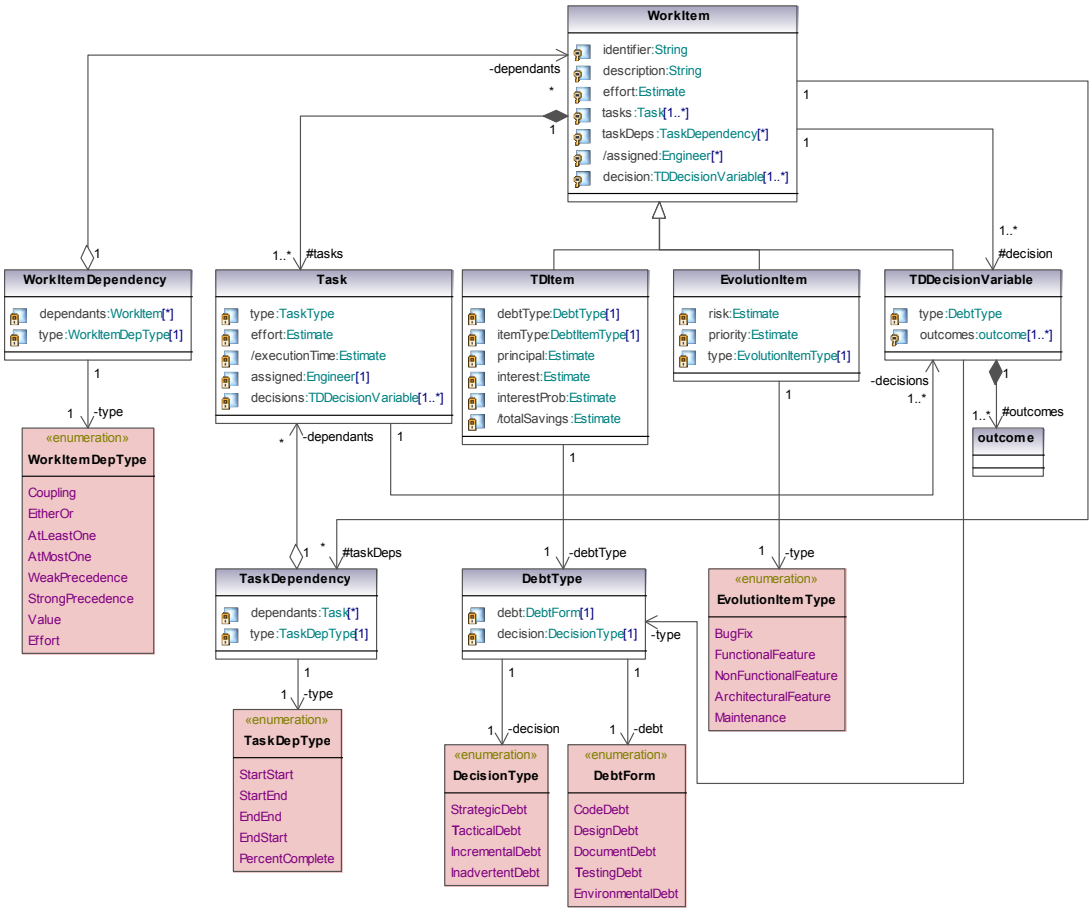

Figure 8. Meta-model section representing the work item, its subtypes and their components.

Work items also have a set of *dependencies* to other work items. This form of dependency was initially identified by Carlshamre et al. [101], when looking at interdependencies between requirements, and further expanded upon by Ruhe [59]. Carlshamre et al. found that typically only about 20% of requirements are independent and that these dependencies must be taken into consideration. Each dependency exists between a set of work items and has a specific *type* which indicates the semantics of the dependency. Currently, the available types of dependencies are as follows:

- *Coupling*(A, B) – both work items, A and B, must be present in the same release due to a bidirectional dependency between them [59].
- *Either Or*(A, B) – a dependency relationship in which only one of the work items, A or B, can be included in the release under consideration [59].
- *At Least One*(A, B, …, n) – a dependency relationship between several work items in which at least one of the items, A..n, must be included in the release under consideration [59].
- *At Most One*(A, B, …, n) – a dependency relationship between several work items, A..n, in which at most one of the work items can be included in the release under consideration [59].
- *Weak Precedence*(A, B) – a dependency relationship indicating that work item A can either be included in the same release as B or in an earlier release, but work item A cannot be included in a release later than the release in which B is included [59]. This also indicates that work item B cannot be included without work item A having been

included in the same or and earlier release, but work item A can be included without work item B being included in any release.

- *Strong Precedence*(A, B) – a dependency relationship indicating that if work items A and B are to be included in the release under consideration, then work item A must be in an earlier release than work item B, and that both work items A and B cannot be included in the same release [59]. This also indicates that work item B cannot be included without work item A having been included in an earlier release, but work item A can be included without work item B being included in any release.
- *Value*(A, B, …, n) – a dependency relationship indicating that when all work items, A..n, that forming the dependency are included in the same release the total value of the set exceeds the sum of the values of the individual work items [59]. That is, there is a bonus increase in apparent value due to the inclusion of all the work items.

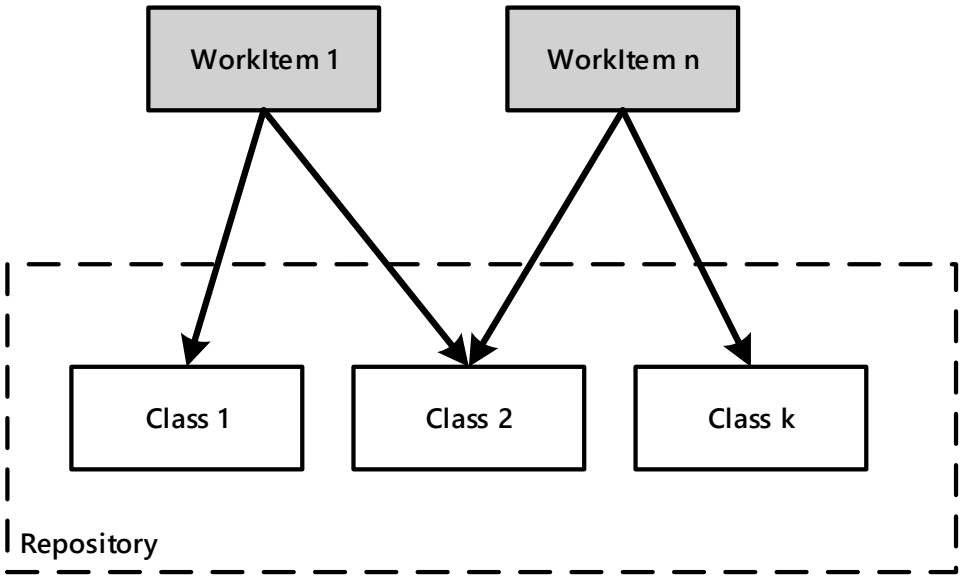

Figure 9. Depiction of architectural dependencies.

- *Effort*(A, B, …, n) – a dependency relationship which indicates that when all work items forming the dependency are included in the same release the total effort of the set is less than the sum of the efforts of the individual work items [59]. That is, there is a bonus apparent reduction in the required effort to implement all items. This could be due, for example, to a refactoring that changes the underlying structure which the remaining work items are extending and reduces the effort to implement these work items.

A second type of dependency contained within work items is the *architectural dependency*, which are depicted in Figure 9. Architectural dependencies represent the connection between work items and a set of program entities on which the work item is dependent or will affect upon implementation. In the context of technical debt items these architectural dependencies are similar to the concept of the *debt flag* extension to the TDMF [26].

Evolution Items. Evolution items represent changes to the System which manifest as observable changes in the behavior, structure, or design of the system. Evolution items have the following specific properties: an evolution *type* which is one of: *bug fix*, *functional feature*, *non-functional feature*, *architectural feature*, and *maintenance*; a *risk* which is the probability that the item will not be completed during its assigned release; and a *priority* which is the weighted average of the priorities assigned by the system stakeholders for this evolution item.

Technical Debt Items. Technical debt items represent identified technical debt within a system and the cost of remediation of these debts. Technical debt items have the

following specific properties: *item type*, *debt type*, *principal*, *interest*, and *total savings*. Item type is one of: *potential technical debt* (sections of the system which have a negative impact on system quality) or *effective technical debt* (potential technical debt that can affect sections of the system which are impacted by current or future evolution items). Debt type is a pair which combines the decision type (*strategic, tactical, incremental,* or *inadvertent*) with the artifact type (*code*, *architecture/design*, *documentation*, *testing*, or *environmental*). The total savings property is an estimate derived from the technical debt matrix for the system [14]. Finally, the principal and interest are estimates used to describe the effort to refactor the impacted areas initially (principal) and growth of this cost over time (interest with associated probability).

Technical Debt Decision Variables. Technical debt decision variables are a type of work item which represent decisions to be made during the software development process in respect to technical debt. The outcomes of these decisions can be to aquire or not acquire different types of technical debt. Thus, each decision variable has the following properties: *type* of technical debt, a set of *outcomes*, and a set of *affected entities*. Decision variables are associated with work items and their tasks, as well as, releases. In the latter case they are used to represent the choices to be made in order to prevent constraints of a release from being violated.

Tasks

Tasks represent the work to be completed during the implementation of its parent work item. Tasks and their components within the meta-model are shown in Figure 8. For example an evolution item representing a new feature would need to be designed,

implemented, and tested. On the other hand a technical debt item would need only be refactored and tested. Each task associated with a work item has a set of *dependencies* to other tasks within that work item. We consider the following types of dependencies between tasks of a work item [40]:

- *Start-Start*: given two tasks, t_1 and t_2, if t_2 has a start-start dependency, β, with t_1, then the start time of t_2, α_2, must be later than the start time of t_1, α_1. That is, if $\beta = \alpha_2 - \alpha_1$, then $\beta \geq 0$. This dependency is depicted in the upper left corner of Figure 10.
- *Start*-End: given two tasks, t_1 and t_2, if t_2 has an start-end dependency, β, with t_1, then the end time of task t_2, ω_2, must be later than the start time of task t_1, α_1. That is, if $\beta = \omega_2 - \alpha_1$, then $\beta \geq 0$. This is depicted in the upper right corner of Figure 10.
- *End-End*: given two tasks, t_1 and t_2, if t_2 has an end-end dependency, β, with t_1, then the end time of task t_2, ω_2, must be later than the end time of task t_1, ω_1. That is, if $\beta = \omega_2 - \omega_1$, then $\beta \geq 0$. This is depicted in the middle right of Figure 10.
- *End-Start*: given two tasks, t_1 and t_2, if t_2 has an end-start dependency, β, with t_1, then the start time of task t_2, α_2, must be later than the end time of task t_1, ω_1.

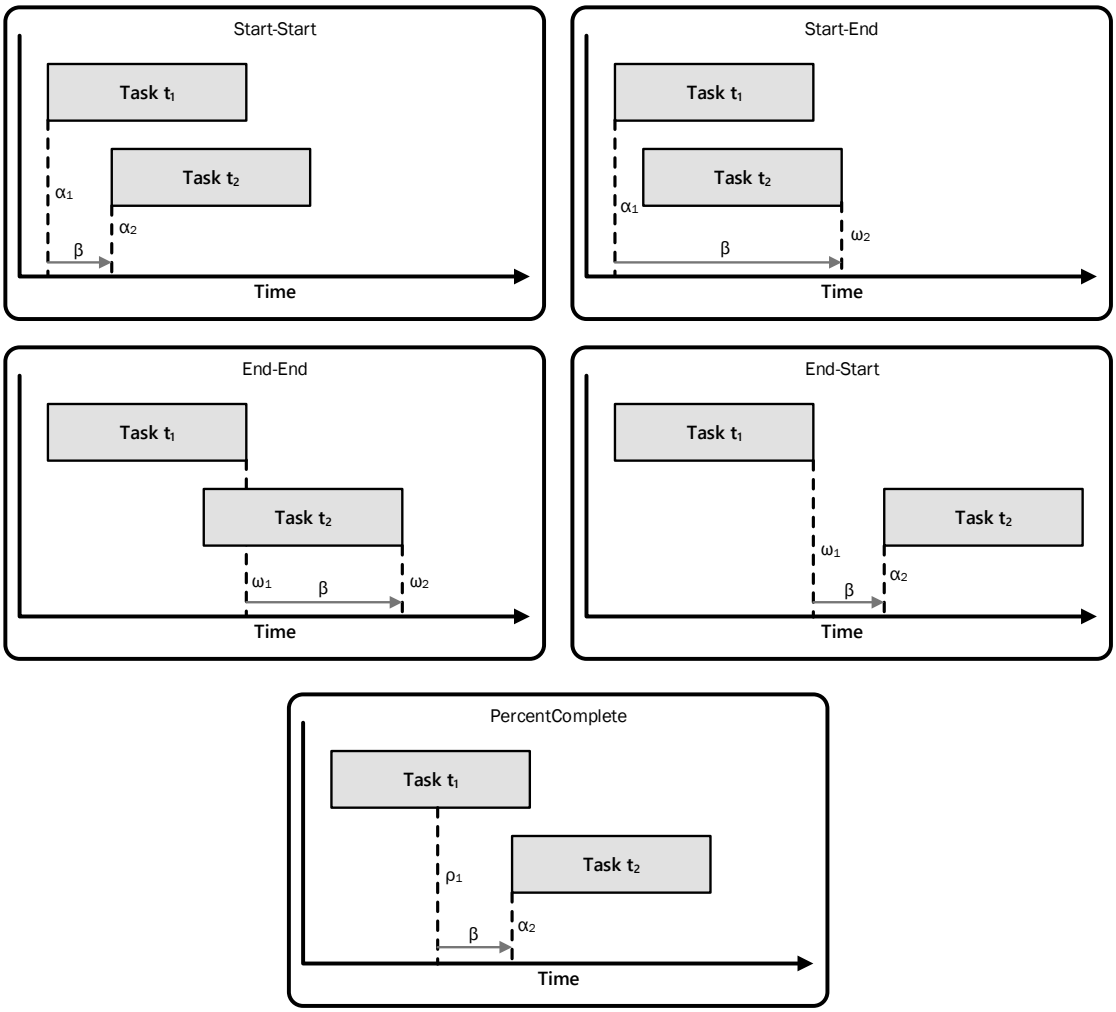

Figure 10. Depiction of the types of dependency constraints between tasks

That is, if $\beta = \alpha_2 - \omega_1$, then $\beta \geq 0$. This is depicted in the middle left of Figure 10.

- *PercentComplete*: given two tasks, t_1 and t_2, t_2 has a x percent complete dependency, β, with t_1, if t_1 must be $x\%$ complete, ρ_1, before task t_2 can be started. That is, if $\beta = \alpha_2 - \rho_1$, then $\beta \geq 0$. This is depicted in the bottom center of Figure 10.

Tasks also have an associated estimated *effort* value. This estimated effort value plays a role in determining the execution time of the task, along with the developer productivity for the task *type* (for the developer assigned to this task). Task *execution time* can be found by dividing the estimated effort required for a task by the assigned developer's productivity for the task type.

Repository

The repository represents the collection of artifacts that define the current state of the system. In the model these are broken down into the following items: Modules, Namespaces, and Program Entities. Each of these items and their properties are defined in the following subsections. The repository and its components within the context of the meta-model are shown in Figure 11.

Modules. Modules (see Figure 11) are the largest unit of a system and represent a single major architectural section of the system. Modules are composed of a set of *namespaces*, and have a *unique identifier* and a *description* identifying its purpose within the system.

Namespaces. Namespaces (see Figure 11) provide a logical division of modules into sections of functionality. Each namespace has an *identifier* which is unique to the module in which it is contained. Namespaces also provide containment for other *namespaces* and *program entities*.

Program Entities. Program entities (see Figure 11) represent the underlying artifacts that implement the system. That is, they represent the classes, methods, and

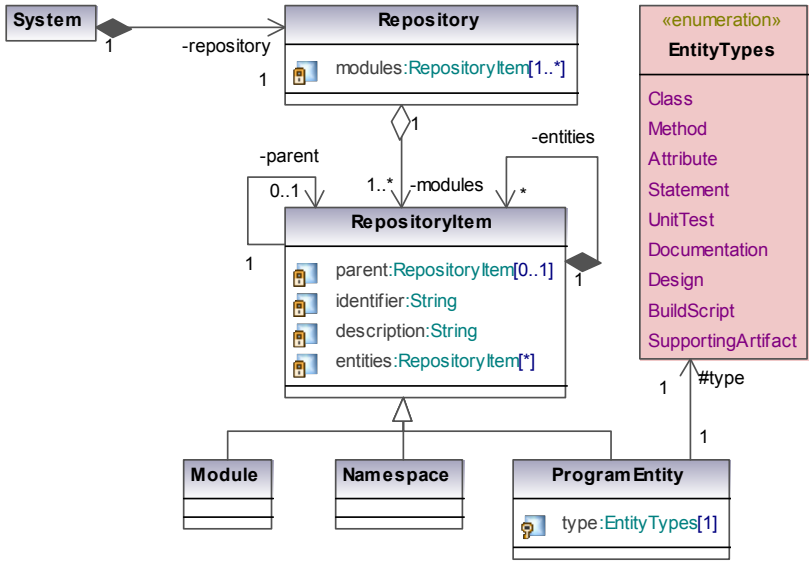

Figure 11. Meta-model section describing the repository and its components.

statements in code, as well as, other artifacts such as unit tests, documentation, and build scripts. Program entities are contained in namespaces and act as containment for other program entities as well. Each program entity has a *unique identifier* (within the scope of its container) as well as a specific *type*: *Class*, *Method*, *Attribute, Statement*, *Unit Test*, *Documentation*, *Design*, *Build Script,* or *Supporting Artifact*.

Product Owners, Stakeholders, and Priorities

Along with the set of stakeholders, the system has a product owner (or manager), which is used to represent the interests of each stakeholder when dealing with software engineers. The manager also acts as the deciding agent (in conjunction with a team lead, if one exists) when dealing with problems such as team configuration or selection of new team members [59]. A depiction of this section of the meta-model is shown in Figure 12.

Stakeholders (including engineers and the product owner) are used to prioritize the items in the evolution sequence and the technical debt list. Although outside the scope of the work in this thesis they still form an integral part of release planning, known as *Prioritization*. Typical issues involved with prioritization involve key stakeholder identification and priority assignment to requirements or work items. In agile software development processes, such as Scrum, stakeholders and their feedback is necessary for the development process [54]. Each stakeholder has a *unique identifier* (i.e., their name or company id) and a *weight* (representing their relative importance).

Prioritization of work items itself has been the subject of recent research. The body of work has surrounded the use of the Analytical Hierarchy Process [80] [102] [103] to perform pairwise comparisons between requirements or features in order to select those which will be a part of the next release [78] [59] [104]. Priority can be one of the following [105] [103] [106] [59] [107]: *Urgency*, *Penalty* (stakeholder dissatisfaction), *Cost*, *Time*, *Risk*, and *Business Value*. Both *urgency* and *penalty* are assigned to each work item by stakeholders and is multiplied by the stakeholder *weight*. Work item priority is the weighted aggregation of stakeholder values which have been normalized to a scale between 0 and 1.

Software Engineers

Software engineers are the agents of change in the system and are depicted in Figure 12. They comprise a team and communicate with each other. Each software engineer utilizes their skills to implement the tasks associated with work items in order to evolve the system. Each software engineer has the following set of properties: A *unique*

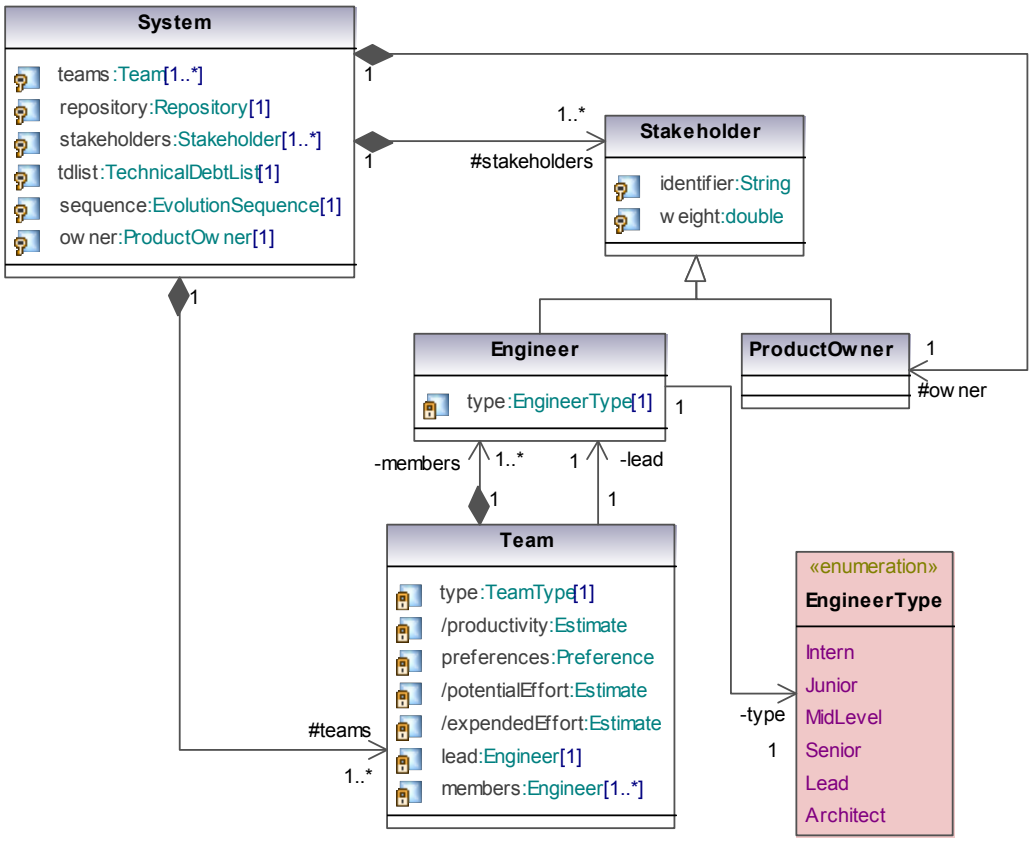

Figure 12. Meta-model section describing stakeholders, product owners, software engineers and priorities.

identifier (the person's name), a *type* (which is one of: *intern, junior, midlevel, senior, architect,* and *team leader*), a *salary*, a *total amount of effort* (effort available per base unit, i.e., in man-hours), *remaining effort* (for the current iteration of development), a set of *preferences* over type of work items, and a set of *productivities* across the types of tasks. The set of productivities are used to represent an engineer's skill level when completing these tasks. Productivities are normalized such that the average engineer on a team has 1.0 for a task type. The set of preferences is used to describe the motivation of an engineer to complete the type of task specified and is on a scale between 0 and 1.

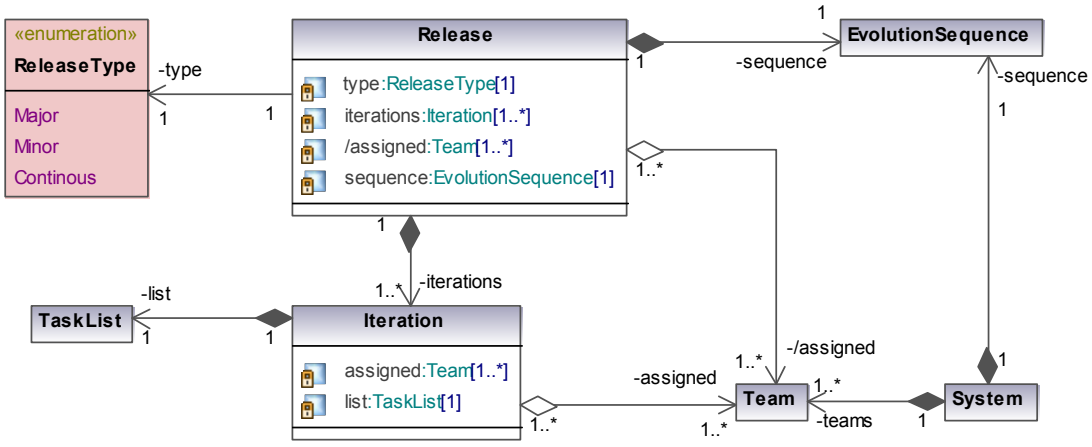

Figure 13. Meta-model section describing releases and iterations and their components.

Releases, Sprints, and Iterations

As a part of iterative (and specifically agile) development processes, the release represents a major development goal. It marks the culmination of development effort which indicates a significant change in a product, such as new features or bug fixes which add value over the last release of a product. The section of the meta-model describing release and iterations is shown in Figure 13. A release is composed of a set of *evolution items* which are to be brought to the user. The release itself is broken down into one or more *iterations* which mark the progress of development. Thus each iteration is a partitioning of the evolution items of a release. Within the framework of the Scrum development process we can insert a further portioning mechanism, the sprint. A sprint is a development period of 45 days of development, working from a sprint backlog (partitioned from the product backlog) [54]. Thus, within this framework a release is composed of one or more sprints which is then composed of one or more iterations.

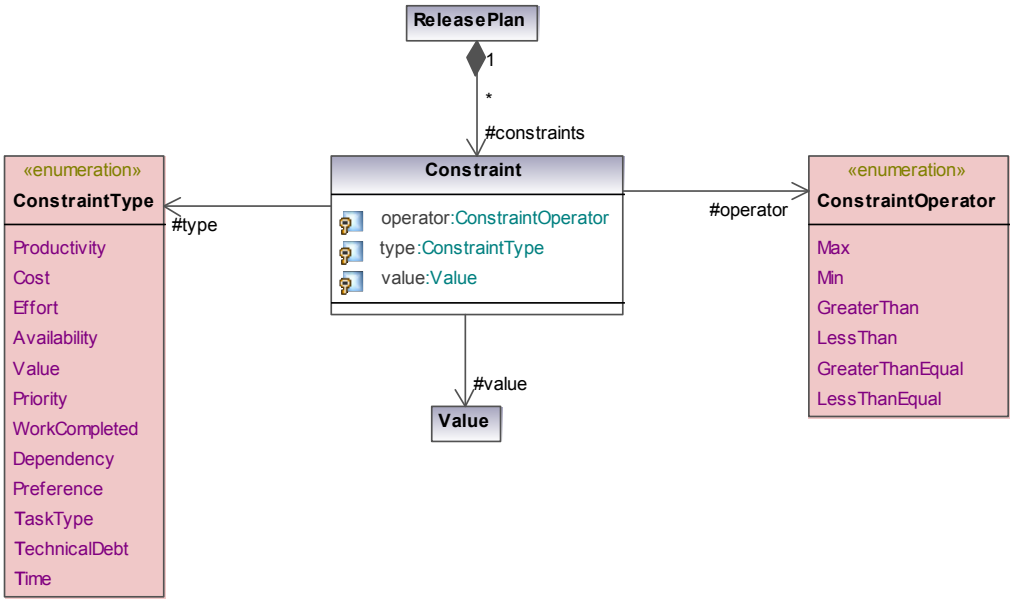

Figure 14. Meta-model section describing constraints.

Constraints

Release plans are subject to a set of constraints as seen in both Figure 14 and Figure 16. Each constraint has an *type, operator,* and may have an associated *value*. The type of constraint determines the items on which the constraint is applied and can be one of the following values: *productivity, cost, effort, availability, priority, value, work completed, dependency, preference, task type, technical debt,* or *time.* The operator is used to define how the constraint is evaluated, we consider the following operators: *max, min, greater than, less than, greater than or equal,* and *less than or equal.* The latter four operators are for threshold constraints.

Estimates, Values, and Probabilities

In this model there are several issues concerning estimation, valuation and uncertainty (see Figure 15). These issues surround the numerical values associated with

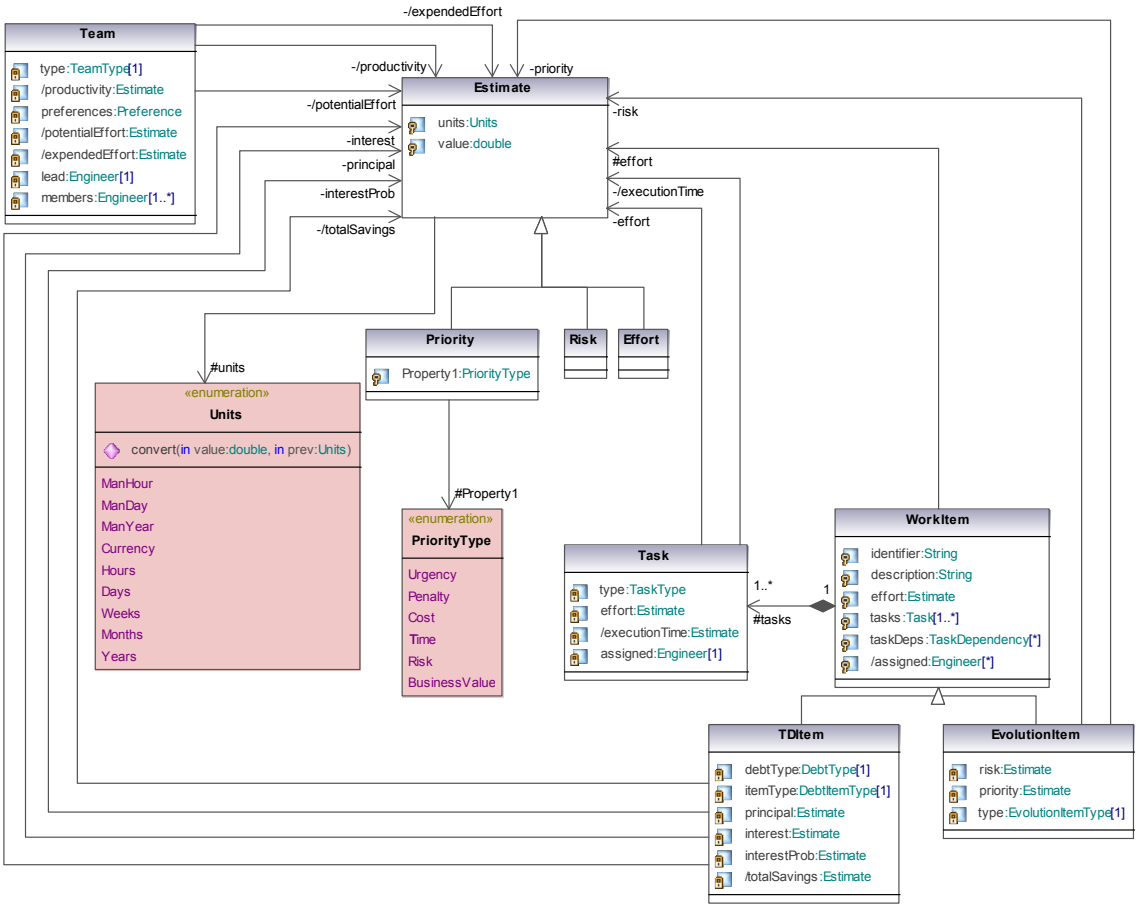

Figure 15. Meta-model section describing estimates and their uses.

effort estimates for evolution items, technical debt items, and the effort values associated with software engineers. These values are only estimates and thus are prone to uncertainty [108]. The value of technical debt as well and its effect on future work along with evolution item value (such as the monetary valuation of these items used in such calculations as net present value (NPV) or return on investment (ROI) [109]) are also estimates and prone to uncertainty. The techniques for the estimation of these values [110] [111] [108] [112] [113] [114] and the assessment of the uncertainty related to these

quantities are important to release planning and technical debt management, but are assumed to be utilized for the purposes of this work.

Release Plans

Release plans are the artifacts generated by release planning methods and algorithms. In this section of the meta-model, depicted in Figure 16, there are two specific release plant types: *strategic release plans* and *operational release plans*. Each release plan is associated with a *system*, a set of *teams*, and a set of *constraints* which the release plan must satisfy. Strategic release plans are composed of *release assignments*, which are a release paired with a set of work items assigned to the release. Operational release plans are composed of a set of *availability constraints* for engineers and a set of

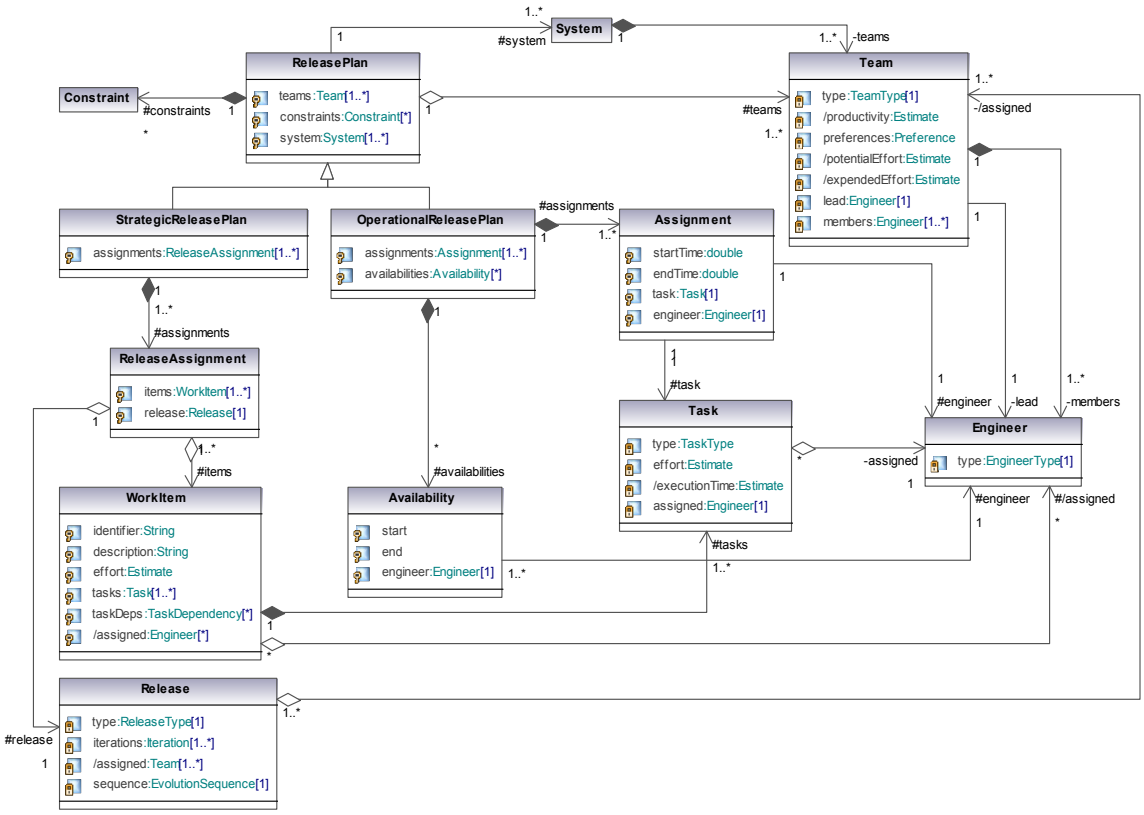

Figure 16. Meta-model section describing release plans.

task assignments for those engineers. Each task assignment is further defined by a *start time*, an *end time*, a *task* and a *software engineer*.

Conclusion

In summary, this chapter combines concepts of release planning and technical debt management into a single coherent meta-model. An implementation of this meta-model provides a means to integrate the concepts of technical debt management into release planning at both the strategic and operational levels, thus allowing for the development of a seamless decision support framework encompassing both areas. This framework then can help managers with decision analysis in order to cope with changes during the development process. In the following chapters we describe initial experiments using reduced forms of this model to explore the effects of technical debt management combined with release planning as well as simulation experiments involving the exploration of technical debt management strategies currently employed by organizations in industry.

COALITION FORMATION GAMES APPROACH

Introduction

This chapter introduces a multilevel solution to address the TDM-SRP and TDM-ORP problems. We consider the problem of the next release and present a solution utilizing a hedonic coalition formation game [88] to distribute work items to teams, and a weighted voting game [115] (one game per team) to assign tasks during the current development cycle. This approach is a promising method to manage known technical debt while still allowing new features to be added to the project backlog.

We use simulation of several teams evolving a project in the presence of technical debt in order to evaluate the effects of this method. The results of these simulations corroborate current thought [5] [116] [7] in the software engineering community with regards to techniques to manage technical debt.

In order to evaluate the approach we compared the hedonic game against a first-come, first-served (FCFS) approach and the weighted voting game against a random assignment approach. We selected FCFS as it is representative of the backlog of agile processes such as Scrum [54]. These comparisons were derived from the following research questions and associated hypotheses:

- **RQ1.1:** Is the distribution of work item effort between teams more similarly distributed (according to cost in units of effort) using Hedonic games than using a FCFS approach?
    - $H_0: \frac{\sigma_{fcfs}}{\sigma_{hedonic}} = 1$. The ratio of the hedonic and FCFS variances is equal to 1.

- $H_A: \frac{\sigma_{fcfs}}{\sigma_{hedonic}} > 1$. The ratio of the hedonic and FCFS variances is greater than 1.

- **RQ2.1:** Does the Voting Game select lower cost items as compared to the FCFS approach?
  - $H_0: C_{voting} - C_{fcfs} = 0$. The difference between the mean cost per selected work item of the voting game and the mean cost per selected work item when using the random game is equal to 0.
  - $H_A: C_{voting} - C_{fcfs} < 0$. The difference between the mean cost per selected work item of the voting game and the mean cost per selected work item when using the random game is less than 0.
- **RQ2.2:** Does the Voting Game select items with a greater gain in benefit (increase in quality or decrease in technical debt) as compared to the FCFS Game?
  - $H_0: B_{voting} - B_{fcfs} = 0$. The difference between the mean benefit (per selected work item) of the voting game and the mean benefit (per selected work item) of the FCFS game is equal to 0.
  - $H_A: B_{voting} - B_{fcfs} > 0$. The difference between the mean benefit (per selected work item) of the voting game and the mean benefit (per selected work item) of the FCFS game is greater than 0.
- **RQ2.3:** Does the Voting Game select items with a higher benefit to cost ratio as compared to the FCFS Game?
  - $H_0: r_{voting} = r_{fcfs}$. The mean benefit to cost ratio for the voting game is equal to that of the FCFS game.

- $H_A: r_{voting} > r_{fcfs}$. The mean benefit to cost ratio for the voting game is greater than that of the FCFS game.

Approach

This approach utilizes cooperative game theory and simulation in order to select which work items and their associated tasks should be completed during each release. Initially, information is gathered to create the project, teams, developers and the associated preferences and properties of each. Once this information is gathered the simulation process begins.

Model

This approach utilizes cooperative game theory [117] to make decisions concerning which work items should completed. Initially, information is gathered to create the project, teams, developers and the associated preferences and properties of each. This information is captured using the following reduced form of the meta-model developed in Chapter 4 and depicted in Figure 17.

Work Items. A work item represents some unit of work that must be done to either build or maintain a software system. For instance it can include the development of a new feature or part of a feature, or it can be the required maintenance of a class or set of classes in the software system's code base. For generality we consider only three types of work items: new features, refactorings, and design defects (technical debt [23]). Each work item w, is formally defined as the following tuple:

$$w = \langle E_w, S_w, \Delta Q, \Delta TD, A_E, A_R \rangle$$

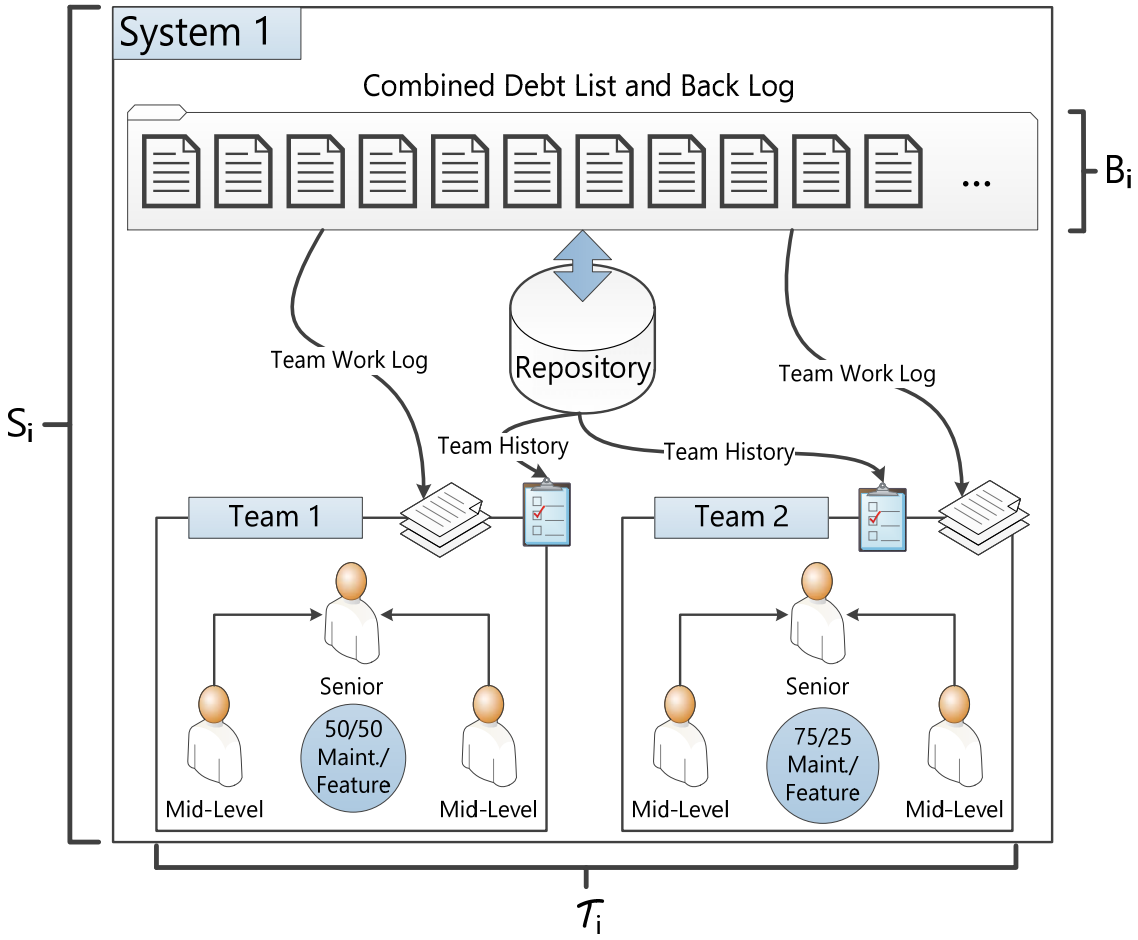

Figure 17. Model depiction of a system.

Where E_w is the effort estimation for this work item, S_w is the size estimation of this work item, ΔQ and ΔTD are the change in quality and technical debt respectively, associated with w, and where A_E and A_R are the set of affected program entities (classes, methods, etc.) and the set of affected regions (packages, modules, etc.), respectively. Effort estimation, E_w is calculated based on relative methods such as using fibonacci numbers, powers of 2, etc. [15] and should be consistent across all work items. The size estimate should be based on some size estimate such as lines of code, number of methods, etc. [15]. Both estimates are parameterizable by the user. Each work item has

two additional derived properties associated with it. The first is the cost of the work item, C_w, derived from the estimates of effort and size of the work item. The second, B_w, is the benefit (increase in quality or reduction in technical debt) realized after completing the work item.

Developers. Teams are comprised of developers; developers complete work items to which they are assigned, and they each are assigned an amount of initial effort per iteration which can be used for the development of new features (including the addition of design debt) or for the maintenance (refactoring) of the current system. The amount of effort is divided according to the developer's preference for these two types of work (where work item type preference is a percentage of the effort they wish to devote towards that type of work). It should be noted that we assume the competent programmer hypothesis, which asserts that programmers are competent and tend to develop programs close to the correct version [118]. The skill of developers is embedded in the model through effort estimations and for each developer an associated probability that a work item will require refactoring. Formally each developer, d, is defined by the following tuple:

$$\mathrm{d} = \langle t, \mathrm{IE}, \phi, \mathrm{t_d}, \mathrm{p_r}, v, w, \mathrm{W_A} \rangle$$

Where t is the team to which this developer is assigned, IE is the effort available at the beginning of each iteration, ϕ is the effort preference set for this developer, t_d is the type of this developer (i.e., Junior, Senior, etc), p_r is the probability that a work item completed by this developer will need to be refactored (this is based on the developer type), v is the number of votes the developer has per work item (during the voting game),

w is the weight associated with the developers selection of a work item (during the voting game), and W_A is the set of work items assigned to the developer during an iteration.

Teams. A team is responsible for some portion of the code base. Each team is comprised of a set of developers and historical record representing the team's experience associated with each module of the code base. The team also has a derived property which is the preferences (maintenance vs. features) used to identify what types of work it will complete (this value is simply an aggregate over the preferences of the developers in D, defined below).

Formally each team, t, is defined by the following tuple:

$$\mathrm{t} = \langle \mathcal{D}, h, \rho, IE, W_A, W_S \rangle$$

Where $\mathcal{D}$ is the set of developers associated with this team, h is the historical record for this team (based on commits to the repository), ρ is the combined preferences of the developers for work item types, IE is the combined sum of the efforts of all developers $d \in \mathcal{D}$ per iteration, W_A is the set of work items assigned to this team and $W_A \subset B_i$ and $t \subset \mathcal{T}_i$ where $\mathcal{T}_i, B_i \in S_i$ (the ith system), and W_S is the set of work items the team selects such that $W_S \subset W_A$. Initially both W_A and $W_S = \emptyset$. Each team also has a defined organizational hierarchy which influences the formal communications between team members and their associated efficiency. This efficiency will affect their initial effort estimations and accounts for errors in estimation. Effort estimation and efficiency penalties are defined by Izurieta et al. [119] and represent the total efficiency penalty for a developer d, E_{T_d}.

Systems. Finally, the model contains the notion of a software system. In the model there is a collection of systems, $\mathcal{S}$. Where the ith member of $\mathcal{S}$, S_i, can be formally defined as the tuple:

$$S_i \quad = \quad \langle \mathcal{T}_i, B_i, R_i \rangle$$

Where $\mathcal{T}_i$ is the collection of teams which are assigned system S_i, B_i is the backlog of work items attached to the system representing undone work such as implementation of new features or required maintenance, and finally R_i is the repository where the system is stored. The backlog B_i is comprised of n work items, $w_1, w_2, \ldots, w_n$. The repository for a system contains P the set of known program entities which are contained in the source code, M is the relationships between entities in P (where entities in the domain are modules, packages, namespaces, or in general, code structures such as classes, methods, fields, etc.), and H, which represents the history of the repository, and is the mapping of developers to committed code over time. A depiction of a system can be seen in Figure 17.

Simulation Process

The simulation process combines the meta-model with coalition formation games in a simulation of the software development process. This simulation process is depicted in Figure 18. Initially we are given a backlog with some number n items corresponding to the total features and maintenance items which need to be completed. During each iteration ($1, \ldots, m$, where m is provided as a parameter) of the algorithm, we take the current back log and distribute the items between the teams (using the hedonic game). Each team then performs a weighted voting game to select work items to be completed.

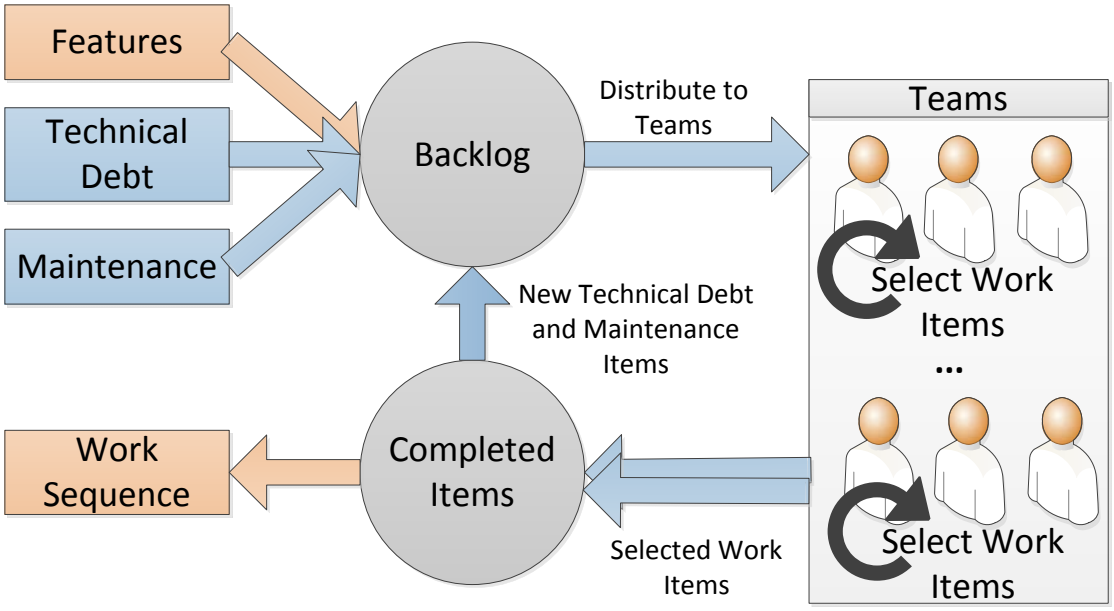

Figure 18. Simulation process.

From this set of selected work items (from all teams), items where $\Delta Q \ll 0$ are considered to be incoming debt items and are converted to refactorings (in a real system these would be detected using design defect detection rules) while features have a slight probability, based on the skill level of the developer charged with implementing it, to be converted to refactorings. The converted refactorings and unselected work items then become the next cycle's backlog and the process repeats until we reach m iterations. The following describes the hedonic and weighted voting games in more detail.

Hedonic Game. The hedonic game is a coalition formation game, $G_H = (N, \phi)$, constructed such that N is the set of all the work items in the backlog B_i and ϕ is set of sets of preferences for each work item for all available team work logs. The team work logs represent the coalitions which work items can join.

Initially the work items are distributed according to a function which minimizes the standard deviation of the total cost of each coalition. Each player then calculates the preference value of its current coalition and a preference value for all other possible coalitions, thus forming a preference profile. Each player then checks their profile for a higher preference coalition that it strictly prefers to their current coalition. If such a

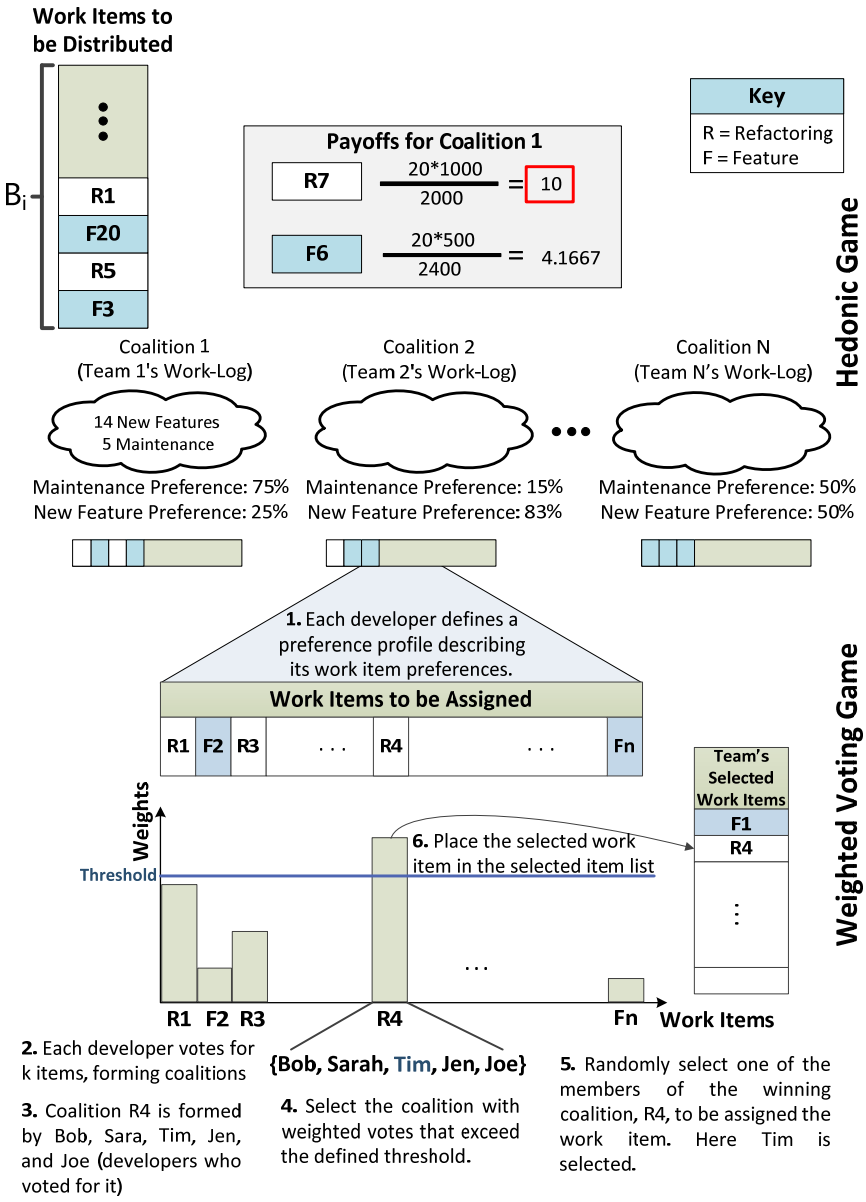

Figure 19. Example depicting both coalition formation games.

coalition is found, the player will update its visitation record to reflect that it has already visited its current coalition and then joins the new coalition. This, in turn, forces all other players to update their preference profiles as well. Eventually all players will settle with a (strictly) preferred coalition. Figure 19 depicts an example of this process.

Weighted Voting Game. The weighted voting games in this approach are each a game, $G_w = (N, \phi)$, and are constructed such that the set N is the set of developers in the team and ϕ is the set of the set of preferences of each developer for each item in the team's work log. In the weighted voting game each developer *n* has the ability to join k coalitions, where k is the maximum number of votes for the developer and each coalition represents a collection of developers voting for a work item (unlike in the previous hedonic game where a coalition was represented by a collection of work items assigned to a team work log). Each vote carries a weight specified by the experience of the developer where more experienced developers have a higher weight. The coalition with a total weighted vote above the threshold is considered the winner, and if several coalitions meet this criterion, then the coalition with the largest number of weighted votes is declared the winner. A random developer selected from the winning coalition is assigned that work item. This item is then removed from the work log and is placed in the selected items list. Figure 19 depicts an example of the weighted voting game.

Methods

This section describes the methodology behind the two experiments we conducted, and it describes the process by which we randomly generate systems for use in these experiments.

Random System Generation

The meta-model presented in Chapter 4 is used to represent the system and its contents, as depicted in Figure 17. In order to evaluate the approaches described above we instantiate several systems randomly. For each of the experiments a number of

Algorithm 1 Random System Generation

$RandSysGen(numSys, numIterations, devTypes, teamDesc, maxSize)$
Input: $numSys$: number of systems to generate,
$numIterations$: number of iterations per system,
$devTypes$: set of developer type descriptions (includes type name, max effort per iteration, number of votes, and voting weight),
$teamDesc$: description of the number of different types and their relationships (within a hierarchy).
Output: $\mathcal{S}$: set of systems.

1. **for** $i \leftarrow 1$ to $numSys$ **do**
2. $TE_{\mathcal{T}} \leftarrow 0$
3. **foreach** $type \in devTypes$ **do**
4. $numMembers \leftarrow |\{x \in teamDesc | x.type = type\}|$
5. $TE_{\mathcal{T}} \leftarrow TE_{\mathcal{T}} + (numMembers \cdot x.IE)$
6. **end foreach**
7. $numWorkItems \leftarrow (TE_{\mathcal{T}} \cdot numIterations \cdot 1.5)$
8. $entities \leftarrow generateProgramEntities(numWorkItems)$
9. $\mathcal{B} \leftarrow generateBacklog(numWorkItems)$
10. $\mathcal{T} \leftarrow generateTeams(numTeams, devTypes, teamDesc)$
11. $\mathcal{S}[i] \leftarrow newSystem(\mathcal{B}, \mathcal{T}, \mathcal{R})$
12. **end for**
13. **return** $\mathcal{S}$

Figure 20. Random system generation algorithm pseudocode.

random systems are generated to which teams are assigned (as defined in Algorithm 1 depicted in Figure 20).

Experiment 1

The purpose of this experiment was to empirically validate that the hedonic game distributes the work items to the team best suited for the task commensurate with the team's preference profile. This is based on the criteria that the distribution of work items should occur to teams with a history of working in the regions of code associated with the item, and such that no single team is overloaded, by keeping the work similarly distributed between participating teams and within the team's effort level. First, we

Table 2. Experimental conditions for experiment 1.

Group	Number of Teams per System
1	10
2	25
3	50
4	100

evaluate whether the game theory approach of distributing work items yields better results than a FCFS assignment mechanism. Furthermore, we evaluate the actual distribution of work items between the teams. A summary of the experimental conditions can be found in Table 2. In this experiment we are concerned with research question RQ1.1 as described in the chapter introduction.

Experiment 2

The second experiment is used to evaluate the voting game approach of distributing work items between developers of a specific team. First, we wish to

Table 4. System characteristics for experiment 2.

Parameter	Value
Number of Systems	100
Number of Teams/System	5
Number of Developers/Team	7
Number of Code Regions/System	15
Number of Iterations	12

determine the usefulness of such an approach versus a random assignment of work items. In order to do this, we divide the system into two sets of experiments. Second, we compare the following two types of models Hedonic-Voting and Hedonic-Random. The first name represents the type of distribution mechanism that assigns work items to teams in a system and the second name represents the method used to select work items for individual developers in a team. The characteristics of the systems for experiment 2 can be found in Table 4.

The experiment was set up using 100 randomly generated systems (generated according to Algorithm 1 in Figure 20). Each system was setup to have 5 teams of 7

Table 3. Developer characteristics for experiment 2.

Type	Votes	Weight	Iteration Effort	Maint./Feat. Ratio
Senior	3	2.0	125	25/75
Mid-level	2	1.5	100	
Junior	1	1.0	75	

developers consisting of the following structure: One senior level lead developer in charge of the team with two mid-level developers each in charge of two junior level developers. The characteristics for each developer are summarized in Table 3. In this

experiment we are concerned with research questions RQ2.1, RQ2.2, and RQ2.3 as defined in the chapter introduction.

Results and Analysis

This section describes the results of the experiments as well as the analysis conducted. Figure 21 shows two plots of mean values recorded during the experiments. The first plot ("Hedonic-Voting vs. Hedonic-FCFS Cost and Benefit") shows the mean

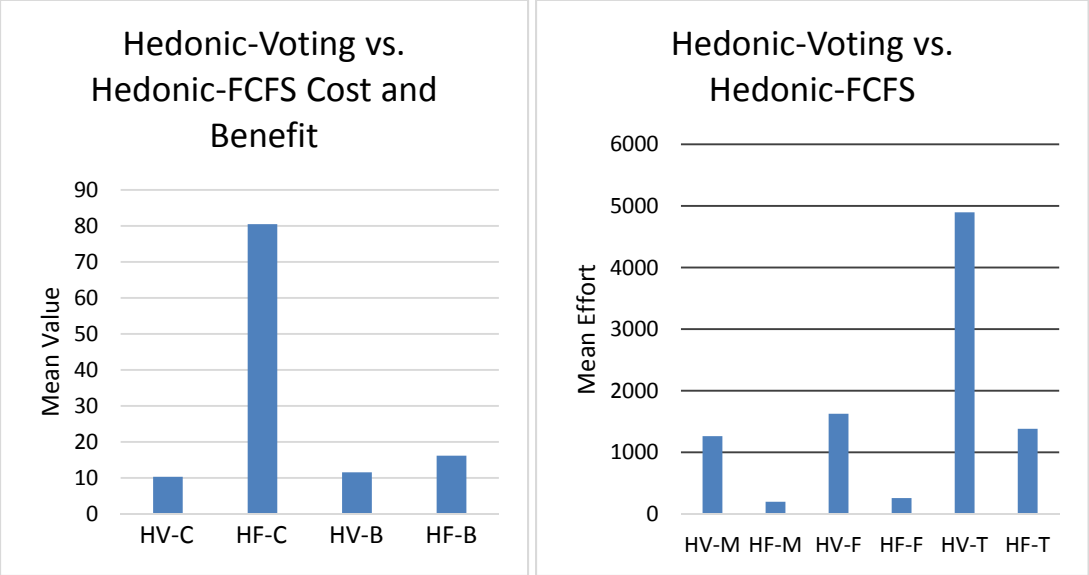

Figure 21. Plots of mean values of each metric during the experiments.

values for cost (-C) and benefit (-B) for the Hedonic-FCFS (HF) and Hedonic-Voting (HV) combinations. The second plot ("Hedonic Voting vs. Hedonic-FCFS") shows the mean effort values for maintenance (-M), features (-F), and technical debt remediation (-T) associated with the Hedonic-Voting (HV) and Hedonic-FCFS (HF) combinations.

Experiment 1

In order to evaluate the evenness of the distribution of work items among teams in a system –hedonic vs. random; we utilized an F-test on the ratio of the variances of the two approaches. This test assumes that the distribution of the sample is normal. In order to assess the normality we utilized Q-Q plots of the data (see Figure 43 of Appendix A). Although there is a slight deviation from normality evident in the plots, the data appears to meet the nearly normal assumption.

Research question RQ1.1 attempts to determine whether the Hedonic Game distributes the work items similarly (according to cost) better than the FCFS Approach. The assessment looked at a ratio of the variance, since if the items are similarly spread between the teams then the variance should be close to zero. Assuming that the Hedonic Game will have the smaller standard deviation the ratio is defined as the standard deviation of the distribution for the FCFS Game divided by the Hedonic Game. We conducted four experiments with 10, 25, 50, and 100 teams to which work items from a backlog would be assigned. The results of the F-tests for each group were all less than 0.1 and thus significant.

Each group in this experiment had similar results leading to the rejection of the null hypothesis in each case. Thus, we can say that there is sufficient evidence to suggest that the true ratio of FCFS to Hedonic variances is greater than 1.

Experiment 2

This experiment was conducted in order to answer three research questions, where each analysis utilized a paired t-test (assumptions verified as follows: the nearly normal

assumption is verified using the Q-Q plots in Figure 44, and the paired data and non-independence assumptions are verified due to the fact that the same systems are evaluated using two separate methods) the findings were as follows:

Research Question 2.1 (RQ2.1). In order to determine whether the Voting Game selects lower cost items as compared to the FCFS Game, we conducted a paired t-test of the difference between mean costs when using the Voting Game versus the FCFS Game. The paired t-test assumes that the data comes from a normal distribution, that the data is paired, and that the data is not independent.

The t-test yielded a p-value < 2.2e-16 which is below the $\alpha = 0.05$ threshold value we have set. Given this result, we have strong evidence to conclude that the mean difference between the mean cost of items the Voting Game selects versus the items that FCFS Game selects is > 0.

Research Question 2.2 (RQ2.2). In order to determine whether the Voting Game selects higher benefit items as compared to the FCFS Game, we conducted a paired t-test of the difference between mean benefit gained. The t-test yielded a p-value < 2.2e-16 which again is below the $\alpha = 0.05$ threshold we have set. Given this we have strong evidence to conclude that the mean difference in benefit between the Voting Game and FCFS Game is significantly greater than 0.

Research Question 2.3 (RQ2.3). Finally, we needed to determine whether the Voting Game selects items with a better benefit-cost ratio than the FCFS approach. In order to determine this we conducted a paired t-test of the mean benefit to cost ratios per

item. The t-test yielded a p-value < 2.2e-16 which again is well below the $\alpha = 0.05$ threshold we have set. This indicates that we have strong evidence to suggest that the voting game is better able to select items with a higher benefit to cost ratio than that of the FCFS game.

All three results show that the Voting Game will select on average lower cost items, yielding a higher overall benefit, and it selects items with an overall larger benefit to cost ratio for the number of items selected. Thus, not surprisingly coalition formation games provide better results than FCFS based assignments.

Analytical Summary

The combined results of the above two experiments not only show that the approach provides a solution to the two problems (TDM-SRP and TDM-ORP1) posed in Chapter 3, but these results also indicate that technical debt can be effectively managed in a project when the following conditions are met:

i. Technical debt items must be included in the release planning stage. As indicated by the model and verified in experiment 1.
ii. Technical debt items must be identified, added to, and tracked in the backlog [5], as indicated by the model and results of experiment 2.
iii. The measures of technical debt must be incorporated as part of the information tracked in the backlog [5] [22] to inform decisions made during operational release planning.

iv. Finally, developers must be willing to actively choose to work on reducing the debt as a part of the maintenance process as well as during the development of features [116].

Finally, although these results corroborate current though in technical debt management, further study and empirical validation is still required.

Threats to Validity

We examine the threats to validity using the Cook and Campbell [120] [121] approach since they are easily mapped to the different steps performed in experimentation.

Construct Validity

Construct validity refers to the meaningfulness of measurements and the quality choices made about independent and dependent variables such that these variables are representative of the theory. If the relationship between the cause and the effect constructs is causal, then the independent variables chosen for the treatment (cause) and the dependent variables representing the output (effect) must be representative of their respective constructs.

In the first experiment we are concerned with how well the variance of total coalition cost accurately measures the evenness of the distribution of the work items between team work logs. Since, the variance measures the deviation from the mean we can conclude that it is a good representation of the effect and that there is little threat to validity. In the second experiment we are concerned with the mean cost, mean benefit,

and the benefit/cost ratio of using coalition formation games over random approaches. These dependent variables are valid, and represent the desired effect construct appropriately.

Several potential threats to construct validity do exist however, and are related to the selection of independent variables for the treatments that represent the cause constructs. The cost associated with each work item is based on relative estimates of size and effort. We can substitute more accurate measures of size such as lines of code and more accurate measures for effort. Further, since this is a simulated environment, many variables are randomly generated. For example, a team's preference, and individual developer's profile, and a system's profile do pose a construct validity threat.

Content Validity

Content validity refers to how complete the measures cover the content domain. The models defining a system and hence the approaches evaluated herein, do not use all known properties of design defects that have been identified by researchers. For example, it is well known that there are dependencies between different defects, which can be used to identify their existence [1] [122] [123] [116], yet these measures are not modeled. We have assumed that a method will be used a-priori to identify design defects in a system; hence these relationships are deliberately discounted.

There is also a potential for refactorings to become conflicted, and for refactorings to be dependent upon each other [124] [7]. The model presented here considers a refactoring entity to be composed of all of its dependent refactorings. We do not take into account the conflicting nature of refactorings.

In order to strengthen the content validity, the models will need to take this into account and the weighted voting game should attempt to utilize conflict information by attempting to select refactorings which reduce the number of conflicts.

External Validity

External Validity refers to the ability to generalize results. Clearly, since this study was conducted on purely synthetic systems (providing face validity [93]) we cannot generalize this approach to real systems. In order to strengthen the external validity of this approach, case studies on a variety of actual software systems is needed. However, the flexibility of the simulations allow for many parameterizations found on already existing parsimonious models or real environments. The ability to vary simulation parameters allows us to describe existing system, team and developer preferences.

Internal Validity

This threat refers to the possibility of having unwanted or unanticipated causal relationships. Since this experiment is fully controlled, this threat does not exist.

Conclusion Validity

This validity check is concerned with establishing statistical significance between the independent variables of the treatments and the dependent variable outputs. Both experiments showed statistical significance in the results obtained. The choice of statistical tests is in-line with the desired hypotheses tests.

Conclusions and Future Work

We have approached the problem of identifying opportunities to remove technical debt such that the selected debt items reflect low cost with potential high benefits. This approach utilizes both a hedonic coalition formation game to divide a backlog of work items between teams and then utilizes a weighted voting game to select the best items from each team such that it meets the team member's preferences. In order to investigate this approach we conducted several simulations using randomly generated software systems. The results of these simulations are encouraging and suggest that further investigation into cooperative game theory as an approach to technical debt management is warranted.

In order to help answer Kruchten et al.'s question [5] of "how to decide about future changes: What evolution should the software system undergo, and in which sequence?," the cooperative game approach presented here moves this line of research in the right direction and provides us with an alternative to improving one of the main issues facing software development organizations today.

Furthermore, the approach shown in this chapter can be further adapted to handle *k* releases and to more formally generate release plans. Currently, we are developing an extension to the base algorithm used in the hedonic game for just these purposes. We are also extending the algorithm to produce multiple Pareto optimal solutions, rather than just one. This work coincides with a combination of the algorithm with the complete meta-model described in Chapter 4. Finally, in the experiments conducted here we explore only the use of a cost-benefit model in order to make technical debt decisions. Since, this

was a part of the foundation to the utility functions, we can easily extend this to be able use Net-Present Value (NPV), Real Options Analysis (ROA), or Total Cost of Ownership (TCO) as suggested by Krutchen, Nord, and Ozkaya [5].

In the future we would like to explore the application of the methods discussed here as a method to explore the connection between technical debt and software evolution. Finally, we are looking into the use of this simulation technique as the basis of a tool to aid in the decision process surrounding system evolution in the face of the technical debt challenge.

INITIAL SIMULATION STUDY

Contribution of Authors and Co-Authors

Manuscript in Chapter 4

Author: Isaac D. Griffith

Contributions: Conceived and implemented the underlying conceptual model. Implemented the simulation models and collected and analyzed the data. Wrote first and final drafts of the manuscript.

Co-Author: Hanane Taffahi

Contributions: Helped validate the implementations of the simulation models. Helped in the analysis of the data. Provided feedback on early drafts of the manuscript and reviewed the final manuscript.

Co-Author: Dr. David Claudio

Contributions: Provided feedback on early drafts of the manuscript and provided review of the final manuscript.

Co-Author: Dr. Clemente Izurieta

Contributions: Helped conceive the study design. Provided feedback and review for the statistical analyses and early drafts of the manuscript. Provided editorial feedback and review of the final manuscript.

Manuscript Information Page

Isaac Griffith, Hanane Taffahi, David Claudio, and Clemente Izurieta
Proceedings of the 2014 Winter Simulation Conference
Status of Manuscript:
____ Prepared for submission to a peer-reviewed journal
____ Officially submitted to a peer-review journal
_x__ Accepted by a peer-reviewed journal
____ Published in a peer-reviewed journal

Will Appear December 7, 2014

Introduction

Technical debt embodies the dichotomy between decisions focusing on the long-term effects to the quality of the software versus focusing on the short term effects on the time-to-market and business value of the software. That is, while software should be delivered on time, any debt (sacrifice in quality) against the quality of the software used to make that possible must eventually be repaid in order to ensure the overall health of the product. This has become a growing concern since as early as 1992 [2], and it was not until recently that industry and researchers worked to provide strategies for incorporating technical debt management into the software development life cycle.

Currently, several basic methods for managing technical debt in practice have been proposed, yet there is little empirical work supporting these claims [28], due to the nature of the problem making empirical studies prohibitive. Thus, simulation provides an excellent alternative to evaluate proposed technical debt management methods, within the context of agile development processes, in a cost and time sensitive way. The problem at hand is to determine, which technical debt management strategy is superior and the most feasible to implement within an existing agile development process model. To investigate the introduction of technical debt management strategies, we have selected the Scrum agile development process [54].

Conceptual Model

The model we have developed is designed to simulate the Scrum development process [54], as depicted in Figure 22, from the perspective of the Product Owner (or manager in charge of a product). In general, the development of the product is done in an iterative fashion, each iteration is called a sprint within which development commences. A sprint typically has a duration of 30 or 45 days, and for this study we selected a sprint duration of 45 days. A release of the software can be composed of several sprints, we

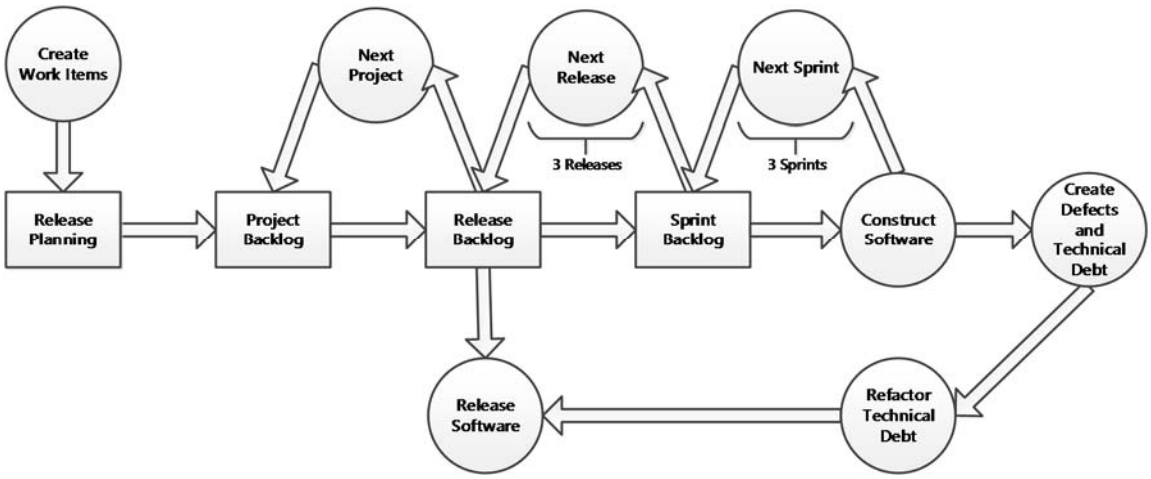

Figure 22. Conceptual model for a discrete-event simulation of the Scrum agile process which includes both defect and technical debt creation.

selected 3 sprints per release for this study. A group of releases then composes a project or milestone for the system. For this study we have selected 3 releases per project. The overall evolution of a system can be decomposed into several projects, but in this study we have limited the number of projects to 1.

The conceptual model consists of three types of objects: Work Items, Software Engineers, and Backlogs. Each work item has the attributes described in Table 5. Each

software engineer has the attributes defined in Table 6. Each of the backlogs consists of the properties defined in Table 7.

Each project begins at the project or release planning stage. This is where the items to be worked on are prioritized and cost and size estimates are provided. Once the estimates are provided the work items move into the project backlog (an ordered list of work to be completed over the duration of the project). This backlog is further subdivided into release backlogs which are further divided into the sprint backlogs. Once a sprint begins the sprint backlog is locked from adding new items until the sprint is complete.

Table 5. Attributes associated with work items in the model.

Attribute	Description
Identifier	A unique identifier to track this work item.
Type	Represents the type of work to be completed and is one from the set {New Feature, Bug/Defect, or Technical Debt (Major Refactoring)}
Priority	A number between 1 and 5 (highest has most priority) and which indicates the desire of stakeholders for the work to be completed. Where a stakeholder is anyone who has a vested interest in the software [136]. Represented as a discrete distribution such that 25% are Priority 1 or Priority 2, 35% are Priority 3, and 10% are Priority 4 or Priority 5. In the case of defects the priority was adjusted such that 50% are Priority 3(1), 35% are Priority 4(2), and 15% are Priority 5(3) for major (minor) defects.
Effort (man-hours)	An estimate of the time it will take for an average software engineer to affect the change to the system. This estimate can be derived from one of many methods (e.g. Planning Poker [108] [112], the Delphi Approach [110]. The effort is set using a triangular distribution TRIANG(0.5, 1, 10), for New Features and Technical Debt, while Defects are set using TRIANG(3,8,24) or TRIANG(1,2,3) for major and minor defects, respectively.
Size (SLOC)	An estimate of the change to the size of the system. The size is represented by a triangular distribution of TRIANG(250,1000,2500).
Engineer	The software engineer assigned to this work item.

1. TRIANG(x,y,z) is the triangular probability distribution, where x is the minimum, y is the mode, and z is the maximum.

Table 6. Attributes associated with software engineers in the model.

Attribute	Description
Type	A representation of the type of software engineer and is one of the following values {Junior, Mid-Level, Senior}. The engineer's type determines their available daily effort and their productivity.
Estimated Daily Effort	An estimate of how much time (in hours) the software engineer has available to put towards working on work items.
Productivity	A factor representing the normalized capability of a software engineer to complete a work item according to that item's estimated effort. The values for the types of software engineers in this model are: • Junior: 2.0 - a junior software engineer takes twice as long as a mid-level software engineer to complete a given task. • Mid-Level: 1.0 • Senior: 0.5 - a senior software engineer takes half as long as a mid-level software engineer to complete a given task.

Once complete the sprint velocity is calculated to determine where the process can be improved. Sprint velocity is a means to determine if the development team was on track when completing the work assigned and provides managers the ability to predict the amount of work a team is capable of handling. Sprint velocity is calculated as the ratio in

Table 7. Description of the backlogs used in the model.

Backlog	Description
Project Backlog	The master list of all work to be completed on the project, and which is ordered using a priority queue. We assume here that the priority also reflects those dependencies between items (or dependencies on artifacts created by the construction of the work items). The product backlog is decomposed into a set of one or more release backlogs as a part of release planning.
Release Backlog	The master list of all work to be completed during a given release period, and it is ordered similar to the project backlog. The release backlog is further decomposed into one or more sprint backlogs.
Sprint Backlog	The master list of all work to be completed during a given sprint, and is ordered similar to the project and release backlogs.

work completed over work assigned between two consecutive sprints. The same metric can be calculated for releases as well as for projects.

At the end of a sprint any incomplete work items are moved from the sprint backlog back into the release backlog. The release backlog is re-evaluated and the next sprint is planned. At the end of each release, the product is delivered to the users. Any remaining work, at the end of a release, is returned to the project backlog. The project backlog is then re-evaluated in order to plan for the next release. The development process continues in this fashion while new work is continually added and evaluated in release planning.

Finally, each newly completed work item can potentially generate new defects (bugs) and/or technical debt. In the case of defects, several processes are typically in place to identify, track, and remediate these issues, yet for technical debt there are typically no such processes in place for technical debt.

The Simulation Process

The general simulation process can be seen in Figure 23 while the input parameters used for each of the models can be found in

Table 8. The following narrative describes this process, utilizing the above defined work items, software engineers, and backlogs.

A release begins by first incrementing the CurrentRelease variable. If CurrentRelease < MaxReleases, then we move items from the project backlog into the current release backlog. Once the release backlog has enough items for MaxSprint sprints (at least MaxSprintEffort amount of work), then the sprint cycle is started. Within the

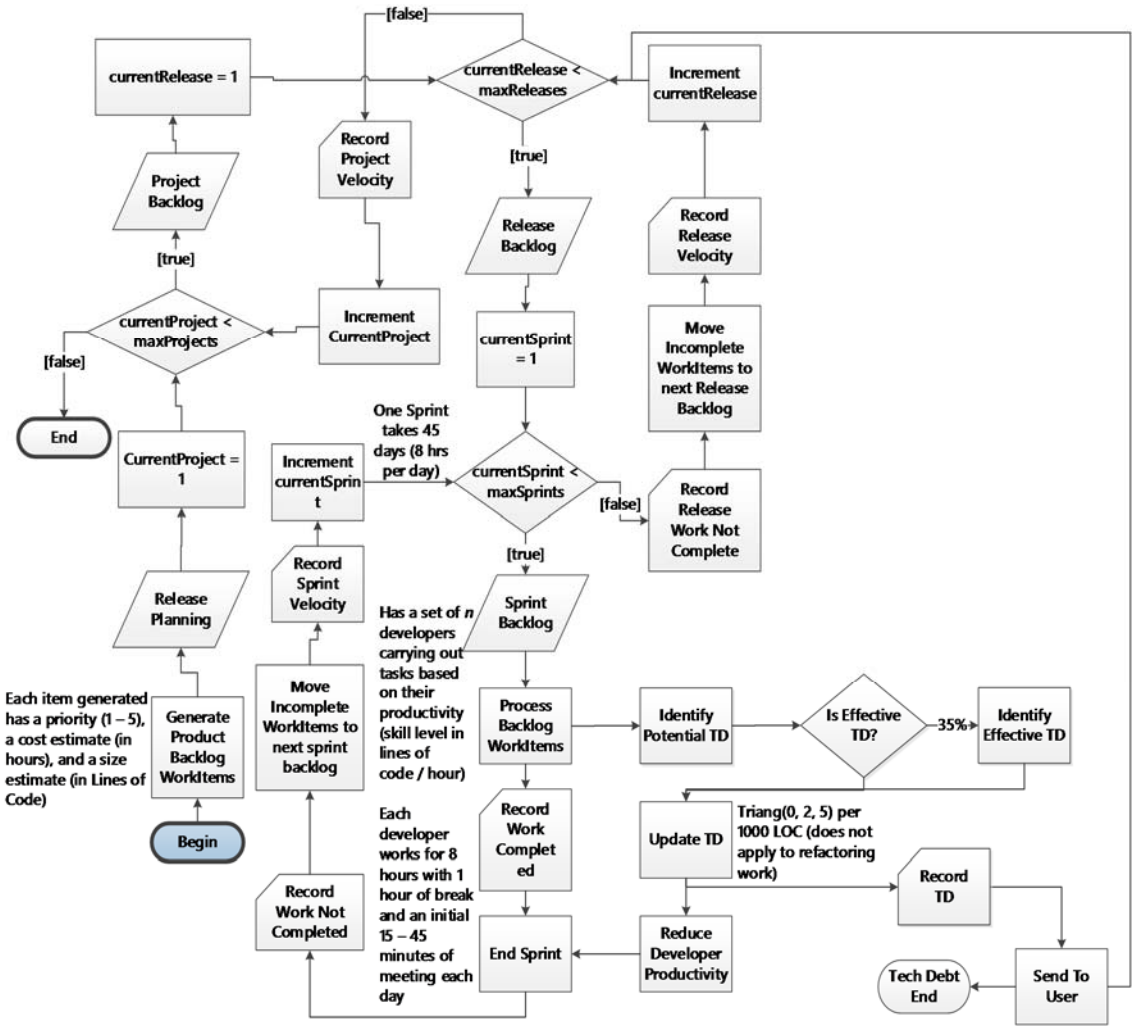

Figure 23. Diagram of the base model for the scrum software development process including defect and technical debt incorporation.

sprint cycle the following occurs: First, the CurrentSprint variable is incremented and then the sprint backlog is filled to capacity (determined by the available effort of the current set of software engineers (MaxSprintEffort)). Once the sprint backlog is filled, work items are then processed by the software engineers. After all items in the sprint have been completed, or the sprint duration has been exceeded, the sprint cycle ends and the next begins. If we have reached the MaxSprints condition, then we start the next

release. If we have reached the MaxReleases condition, then we begin the next project. Finally, if we have reached the MaxProjects condition, then we end the simulation.

During each sprint, as the software engineers are completing the work items, it is possible that each completed work item will generate potential technical debt. The work items are still considered complete but at the same time the model generates new technical debt items for processing. The simulation generates TRIANG(0, 2, 5) number of new technical debt items per 1000 SLOC. In the base model, the technical debt items are not tracked or actively identified and thus leave the system as a part of the production

Table 8. Input parameters, their descriptions and default values used during simulation.

Input	Description	Value
$MaxSprintEffort$	Maximum effort assignable to a sprint.	1800 man-hours
$MaxReleaseEffort$	Maximum effort assignable to a release.	5200 man-hours
$MaxProjectEffort$	Maximum effort assignable to a project.	16200 man-hours
$MaxSprints$	Maximum number of sprints per release.	3 sprints
$MaxReleases$	Maximum number of releases per project.	3 releases
$MaxProjects$	Maximum projects per simulation.	1 projects
$InitialTD$	Initial amount of TD in the system.	1000 SLOC
$SprintDuration$	Maximum sprint length in days.	45 days
$SprintTDIteration$	Number of sprints between TD-only sprint occurrences.	2 sprints
$SprintTDPercent$	Percentage of sprint effort dedicated to TD.	15%
$SystemSize$	Initial size of the current system.	8500 SLOC
$TDLowerThreshold$	Minimum threshold for TD.	1000 man-hours
$TDUpperThreshold$	Maximum threshold for TD.	5000 man-hours

product. It should be noted that for the technical debt generated we are counting the identified (for models where active tracking is used) and unidentified (for all models) instances as variables of the system. We specifically track technical debt, as a part of the simulation (not to be confused with the technical debt list), to impose a penalty on software engineer productivity as shown in (1). The argument for this reduction in productivity is based on the notion that technical debt embodies the impact of poor quality on the cost of change to a system. Thus, if the cost of change increases while the number of software engineers stays constant, the impact is that their productivity (ability to affect the change on the system) must be decreasing, as defined by the following formula:

$$DeveloperProductivity = \frac{1}{1 - \left(\frac{TechDebtSize}{SystemSize}\right)} \quad (1)$$

This conceptual model assumes the following is true: The stakeholders and product owner have assigned priorities to each of the work items with a value between 1 and 5. The new features to be developed have been decomposed into the smallest workable units. In the base model, we assume that technical debt is not a concern and that any refactoring is not intended to remove technical debt. We assume that release re-planning occurs but is outside the scope of these models. We assume that the estimates for cost and size are correct. Finally, we assume that the priority of the work items and their order in the list also reflects the dependencies between them. That is, if a work item is dependent upon other work items, then those it depends upon are listed before it in the backlog.

Experimental Design

This section outlines the experiments and data generation methods used in conducting this simulation study. We first describe the experiments conducted and then describe the data generation procedure.

Experiments

The experiments are designed to explore different methods of technical debt management which have been proposed in the literature. Specifically we have identified four models which are used for comparative analysis. The models have been developed in a hierarchical fashion, with each adding new features on top of the previous model. The base model (Base) is an implementation of the conceptual model, does not consider technical debt management, and is used to verify that the process is correct prior to evaluating the other approaches. The second model (TD List) maintains a separate list of technical debt items which allows for deliberate tracking of the technical debt items. The remaining two models use this list and continuously monitor development of new instances of technical debt.

These two models, TD List and TD List with Active TDM, can use either a percentage based or sprint based strategy to remove technical debt. In the percentage based method, a certain percent of sprint effort is directed toward the removal of technical debt while the rest is directed toward defect or new feature work. In the sprint-based method, every nth sprint's entire effort is directed toward the removal of technical debt. The final model is based on the concept of a technical debt threshold [17], which is built upon the active monitoring model and utilizes a threshold to identify when technical

Table 9. Summary of the models and strategies developed for comparative analysis.

Model	TD Remediation Strategy	Simulation
1. Base	-	**Base**
2. TD List	Percent Sprint	**TDL-P** **TDL-S**
3. TD List with Active TDM	Percent Sprint	**ATDM-P** **ATDM-S**
4. TD Thresholding	Upper Threshold Only Upper and Lower Threshold	**TDT-U** **TDT-UL**

debt should be removed. This model has two possible threshold approaches: the first begins technical debt removal once the current level has reached an upper threshold, and the other utilizes both an upper threshold a lower threshold to stop the technical debt removal phase.

Using these models we construct and compare the results of each simulation and the various strategies employed in order to determine which technical debt management strategy is superior. First, we compare between strategies of each model, then we compare between model types using the best alternative at each level for the *between-level* comparisons. In each of these comparisons we look at the following five metrics: *cost of completed items* (CC), *count of work items completed* (WC), *cost of effective technical debt* (ETD), *cost of potential technical debt* (PTD), and *cost of total technical debt* (TD). For CC, ETD, PTD, and TD each is measured in source lines of code (SLOC). Each of these values are mean value for a single simulation run averaged across all of the repetitions of the simulation. A summary of these models can be found in Table 9.

Data Generation

Utilizing existing theoretical concepts and models we randomly generate new features, technical debt items, and defect items, using the distributions previously noted. The generated features will have sizes and effort estimates corresponding to values that would be achieved using the methods identified in [111] and [108]. The size and cost/effort estimates for technical debt items are based on the models identified in [125], [36], and [10]. The defects generated during the process follow the empirical models described in [126] which identifies the size and estimated effort required to remove these defects.

Results and Analysis

We conducted several simulations of the models described in the previous section. For each simulation we conducted a total of 8125 replications. The number of replications was selected in order to reduce the percent-error of the metrics of concern (most notably TD) to within a half-width of 1.5%. The resulting average of the mean

Table 10. Average differences for each metric from each comparative analysis.

Comparison	CC (SLOC)	WC (Count)	ETD (SLOC)	PTD (SLOC)	TD (SLOC)
TDL-S vs TDL-P	117.956	-9.544	14.164	-13.552	65.518
TDL-P vs Base	-2536.8	1393.292	-528.332	-2310.236	-1921.887
ATDM-S vs ATDM-P	-645.264	350.604	-137.416	-617.648	-556.348
ATDM-S vs TDL-P	-548.724	420.008	-105.564	-506.408	-462.038
TDT-U vs TDT-UL	2662.508	-1369.668	548.176	2325.976	1959.104
ATDM-S vs TDT-U	-2565.968	1439.072	-518.784	-2220.152	-1869.517
TDL-P vs ATDM-P	125.708	23.624	19.844	15.74	37.217

metrics values for each metric of concern over the developed models can be found in Table 6. Figure 24 depicts the mean metric values (excluding WC) between simulations, while Figure 25 depicts the change in CC, WC, and TD across simulations. Each comparison, whose values are shown in

Table 10, was conducted using a two-tail t-test ($\alpha = 0.05$). In the comparison between the sprint-only and percentage based TDM strategies on the TD-List method, we found

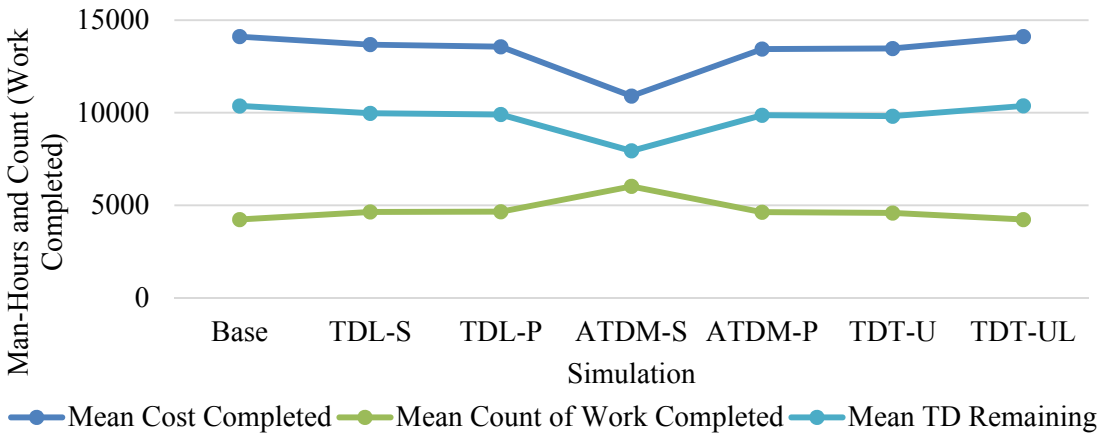

Figure 25. Change in work completed, technical debt remaining and mean cost completed across simulations.

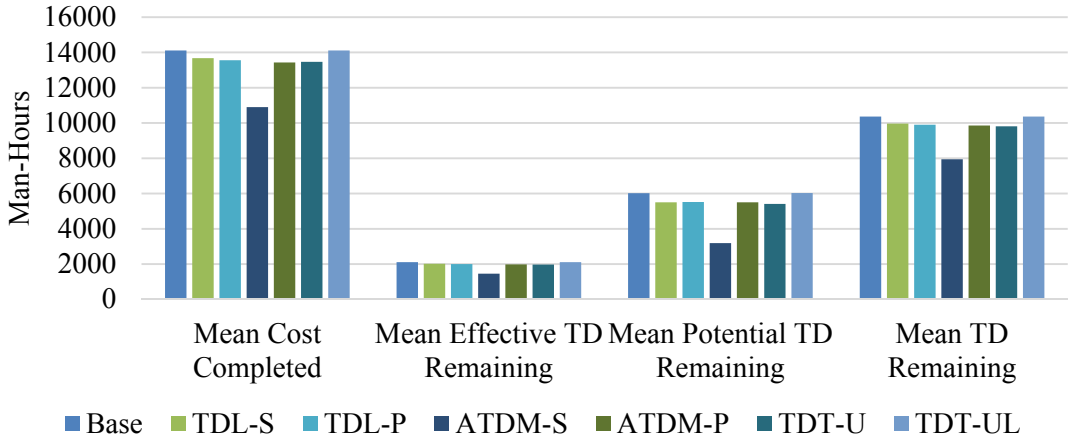

Figure 24. Comparison of metrics across simulations.

that the percentage based approach was superior. The reasoning behind this is that the percentage based results showed that more work items where completed at a reduced cost, while more technical debt (specifically effective technical debt) was removed. Using these results we then conducted a comparison between the percentage based technical debt list combination and the base model (no TDM). Here, not surprisingly, we see similar results, in that the percentage based technical debt list combination removes more technical debt and completes more work items at a reduced cost.

In the second set of comparisons we began by looking within the technical debt list with the automated TD monitoring method. Here, we compared the sprint-only and percentage based approaches. To our surprise, and contrary to the literature, the sprint-only method was found to be superior. This indicates that the sprint-only approach completes more work for less cost but also reduces technical debt (both potential and effective technical debt) better than the percentage-based approach. We note that while the sprint-based automated TD monitoring approach is superior to its percentage-based competitor, in practice this is not necessarily feasible due to such concerns as time-to-market or developer morale (which are not considered in these simulations). We then compared both approaches to the percentage based technical debt list combination. The results indicate clearly that the sprint-only automated TD monitoring combination was superior. As for the percentage based automated TD monitoring the results showed that although this approach does remove more technical debt than the technical debt list only combination, it completes less work.

The final set of comparisons began by comparing the automated technical debt monitoring approach with two thresholding strategies. In these comparisons we found that the use of an upper limit threshold is superior to a ranged threshold and reduces the technical debt and effectively completes more work in a more cost effective manner than a combined upper and lower threshold scheme. When comparing the upper threshold strategy to the sprint-only strategy from the previous set of comparisons, we found that the sprint-only strategy was superior. This result comes with a caveat, in that, in order to further validate this result, sensitivity analysis needs to be conducted in order to both identify the best thresholds and to identify how the thresholds actually affect the simulation. A similar sensitivity analysis needs to be applied to both the percentage based approaches and to the sprint-only based approaches.

Conclusion

We described a set of models representing several different technical debt management methods and their combinations. The context of this study was set in a model of the agile development process known as Scrum. Our study shows that combining a prioritized list of technical debt items in parallel to the development backlog, while continuously monitoring for both known and unknown technical debt items and focusing either a percent of sprint effort or all of every *n*th sprints effort on technical debt remediation sprints is the superior combination of practical technical debt management technique. This result provides empirical support for several of the basic strategies for managing technical debt that have been recently put forth in the literature. Yet, it brings into question earlier notions that development teams cannot stop new

feature work to only focus on technical debt. As noted earlier, this surprising result may be attributed to the fact that we did not take into consideration such things as developer morale and time-to-market concerns.

It should also be noted that we did not try all combinations due to time constraints and that using thresholds may still prove a viable technique. In future work we intend to continue to explore various combinations as well as conduct sensitivity analysis on the various parameters associated with the simulation (see Table 1). We are also looking to combine these models with more advanced approaches to technical debt management as a means to evaluate how the addition of decision support can help effect more efficient technical debt reduction while ensuring continual feature development. A final note on future work is that once the sensitivity analysis is complete we will begin validation of the model using data from several open-source and potentially industry projects.

AN EXTENDED SIMULATION FRAMEWORK

Introduction

The simulation model presented in Chapter 6 provided a means to evaluate technical debt management techniques currently in use in industry. It also provided an initial model upon which a simulation framework for decision support can be developed. In this chaper we present current work towards such a simulation, which utilizes the conceptual model developed in Chapter 4. This simulation framework has been designed to connect the meta-model with the algorithms developed and demonstrated in Chapter 5 (along with other well known approaches from the literature) in order to solve the problems identified in Chapter 3. Along with a simulation framework this chapter identifies the types of questions which can be addressed and methods for sensitivity analysis. For the latter we provide example experiments which we leave to future work.

Simulation Model

In order to provide decision support for release planning and technical debt management we have extended the simulation model defined in chapter 4. This section describes the conceptual model of the enhanced simulation model, the detailed processes underlying the simulation, and the parameters controlling the simulation. The following section describes the conceptual model underlying the simulation framework.

Conceptual Model

The conceptual model is updated to consider both software engineers and work items as entities. We have also incorporated tasks associated with work items as part of

the simulation. Finally, we have abstracted the simulation process to be able to handle generic iterative software development processes beyond Scrum.

The model begins with the creation of software engineers and the creation of work items and tasks. Once created the software engineers are placed in the engineer pool to await the start of the development phase process. Similarly when work items and tasks are created they either enter the project backlog (evolution items) or enter the technical debt list (technical debt items). The tasks that are created are kept with the work items until the Minor Release Planning process. Along with the creation of these entities, the model is initialized with its parameters.

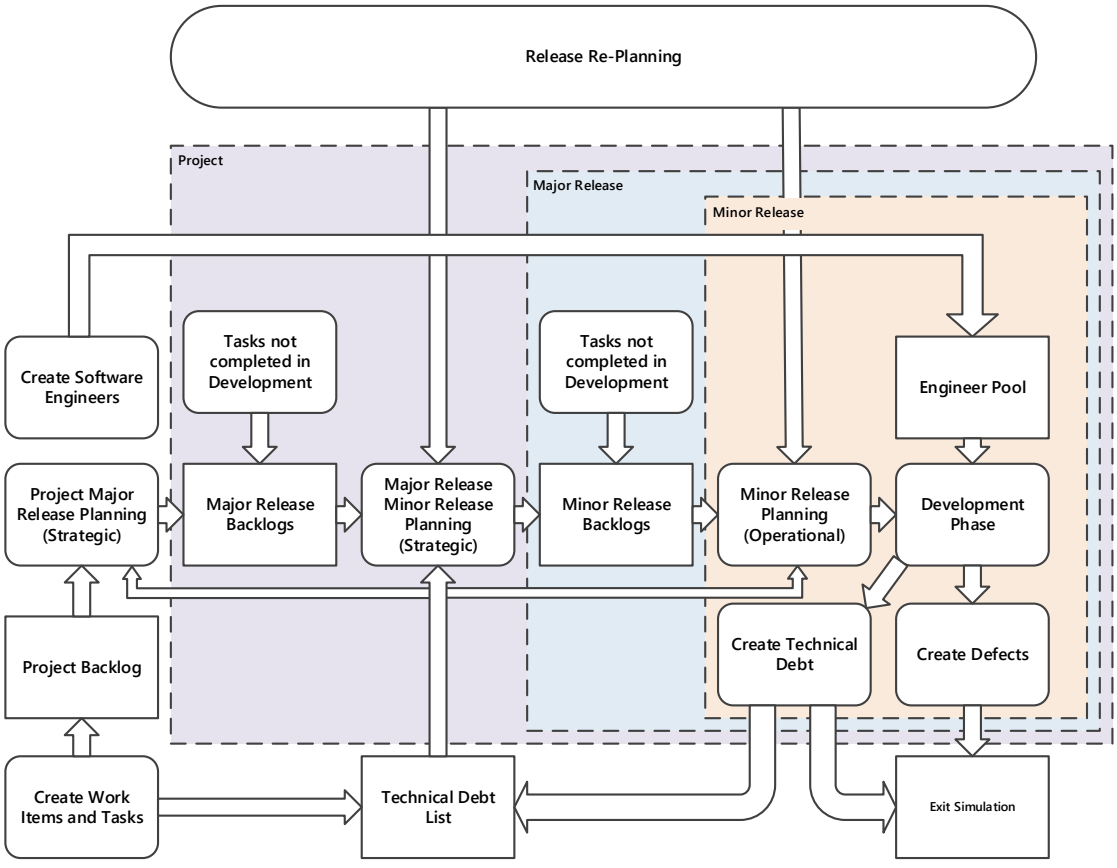

Figure 26. Conceptual model of the discrete-event simulation component.

Once model initialization is complete, Project Release Planning occurs. In the Project Release Planning process work items are distributed to the major release backlogs. Once the initial distribution occurs, each major release begins. After each major release, any tasks (and their associated work items) are moved into the next Major Release Backlog (evolution sequence). Within each major release the next process is the distribution of items between minor releases.

The Minor Release Planning process uses a strategic release planning process to schedule which work items (both evolution and technical debt items) will be completed during the minor releases of the current major release. This process creates a portioning of the current major release's backlog into minor releases. This then transitions the simulation into the minor release section.

The minor release phase of the simulation is where all of the work is performed. It begins by portioning the minor release backlog into an operational release plan. This is performed using an operational release planning method to generate a release plan which includes assignment of tasks to engineers such that all the applicable constraints are met. At this point, the development phase begins.

The Development Phase process follows the schedule set out by the operational release plan. As time progresses the phase selects the next task and engineer pair and places them into a current development queue to wait until the item is completed. Depending on whether the goal is to minimize cost over a fixed time or to minimize time with fixed resources the process will continue. In the former case, the simulation will halt the development phase when the minor release date is reached. In either case any tasks

not completed during the development phases will be moved to the next minor release. Once a work item/task is completed it moves through the defect generation and technical debt generation processes. The technical debt generation process will potentially move new technical debt items into the technical debt list, but the defect generation process only records information. In both cases, the completed work item/task exits the simulation after final processing.

The remaining process, Release Re-Planning, involves the modification of the simulation in order to evaluate the effect of changing circumstances. This process allows the manager to use the simulation to adjust variables such as developer productivities, effort estimations, technical debt thresholds, etc. The re-planning process allows users of the simulation to evaluate how the changes will affect the plan.

Simulation Process

The following subsections describe in detail the extended simulation model's processes as defined in the conceptual model. Each of the processes have been defined using UML[1] activity diagrams.

Work Item Generation and Lifecycle. Each work item has two main processes associated with it. The first is the work item generation process (as depicted in the

[1] http://www.uml.org

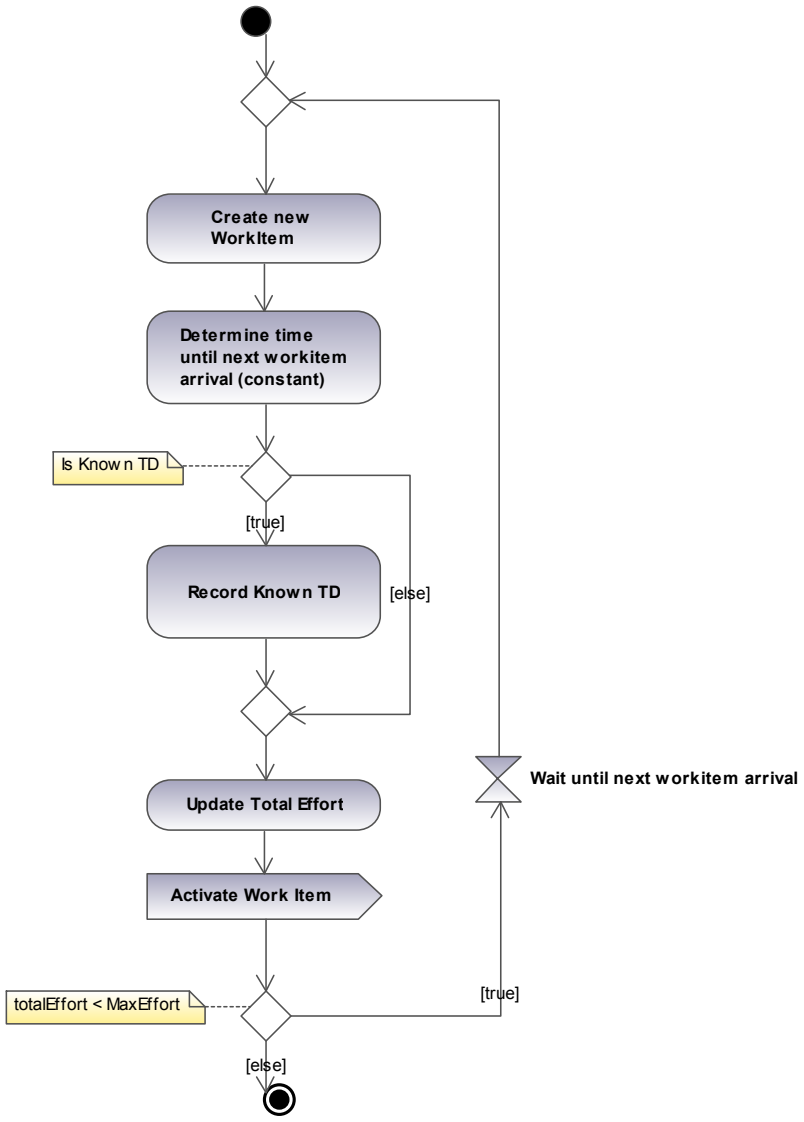

Figure 27. Work item generation process activity diagram.

activity diagram in Figure 27). The process starts by creating a new work item (using information gathered from external sources). Once the work item is created the process calculates the time of the next work item arrival (creation event), which is a small constant value. If the newly created work item is a technical debt item, the value of known technical debt is increased. The total project effort is then updated using the

estimated effort to complete the work item. After the total effort is updated, the work item is then sent to the project backlog to await project planning. At this point the newly created work item is activated and the work item lifecycle process begins. Finally, if the total effort for the project is less than the maximum effort available (if using specific release dates) or there are still work items remaining (no specific release dates), then the process waits until the next work item arrival event occurs and continues the process.

The second process associated with a work item is its actual lifecylce (depicted in the activity diagram of Figure 28). Initially, the work item generates the tasks associated with it. Once the tasks have been generated it begins waiting for the next set of events. There is a possibility of one of five events to occur for any given work item: *a change in*

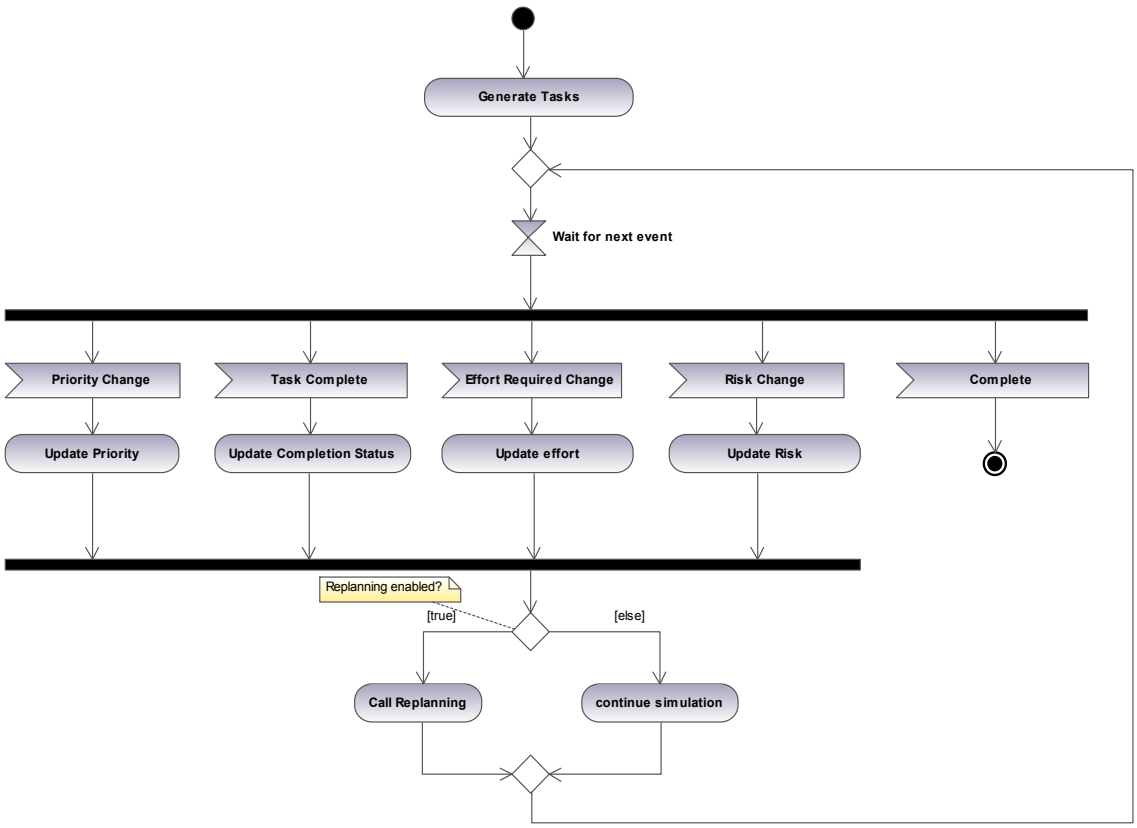

Figure 28. Work item lifecycle activity diagram.

priority, an associated task is completed, a change in reequired effort estimate, a change in the risk estimate, or *the work item has been completed*. The first four of these events cause the work item to update its internal state (priority, completion status, effort required, or risk). Once any of these events has been processed replanning or continuation of the simulation occurs. If replanning is enable and enough of a change in the project has occurred to trigger replanning, then replanning will occur, otherwise the simulation is continued. The work item lifecycle continues until a complete event occurs which indicates the end of life for this work item.

Task Generation and Lifecycle. Each task is generated as part of the work item activation (see Figure 29). The generation reads in the necessary state information from

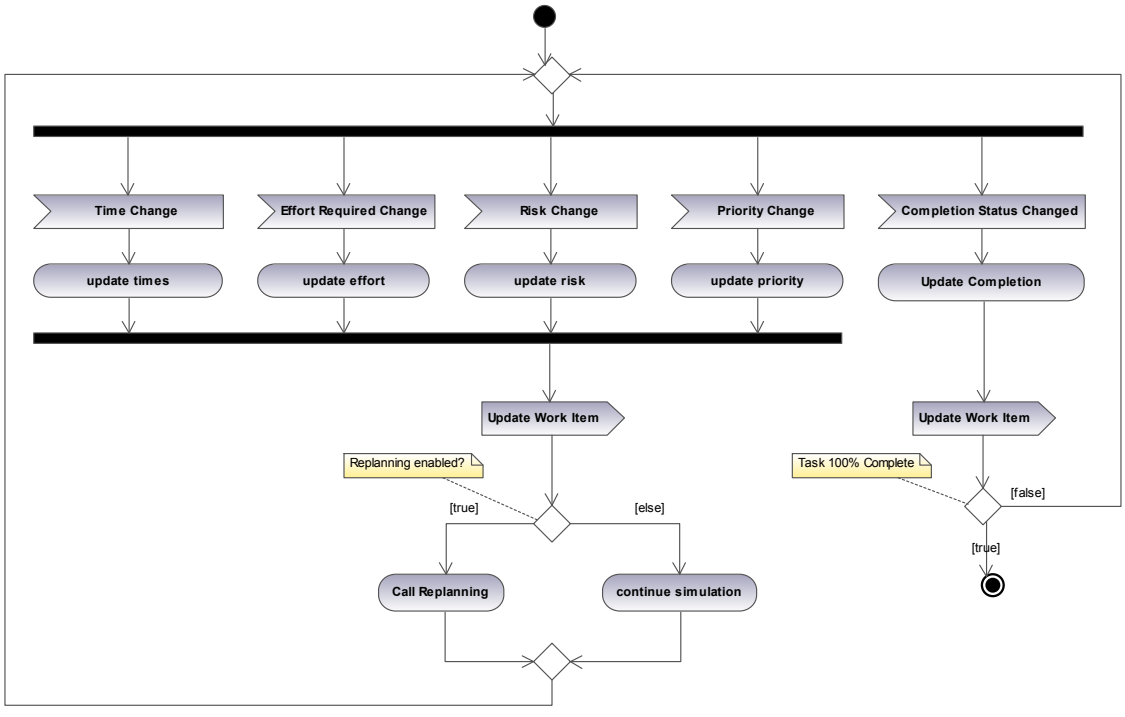

Figure 29. Task lifecycle activity diagram.

an external source such as a database or requirements management tool. Once the tasks are created they are then activated. This activation signals the lifecycle of a task.

The task lifelycle is depicted in diagram. The entire lifecycle is spent waiting for one of the following events to occur: *time change*, *effort required change*, *risk change*, *priority change*, *completion status change*. When any of these events occur the task updates its internal state and then signals the associated work item to update its state. Each of these events, excluding completion status change, can cause re-planning to occur, if re-planning is enabled. If not enabled, the simulation just continues. If an completion status update occurs, the lifecycle only continues if current percentage complete is less than 100%, otherwise the task lifecyle is terminated.

Defect Generation. For any task that has been completed in the development phase process there is a chance that it will contain defects. Thus, the defect generation process, as depicted in Figure 30, begins by a calculation of the time for next arrival of a Task from the development phase. The process then waits until this occurs and uses the task to generate the defects. Once the defect generation process has occurred two simultaneous operations occur. One is the sending of newly created defects to the project backlog, the other is the termination of the completed task.

The actual generation of defects from a task is depicted in Figure 31. In this process there is a 85% chance that a minor defect (takes between 1 and 3 man-days to correct) is created and a 15% chance that a major defect (takes between 3 and 5 man-days to correct) is created. At this point there is a 65% chace that the newly created defect will be caught in the current minor release's testing phase, which will then place the defect

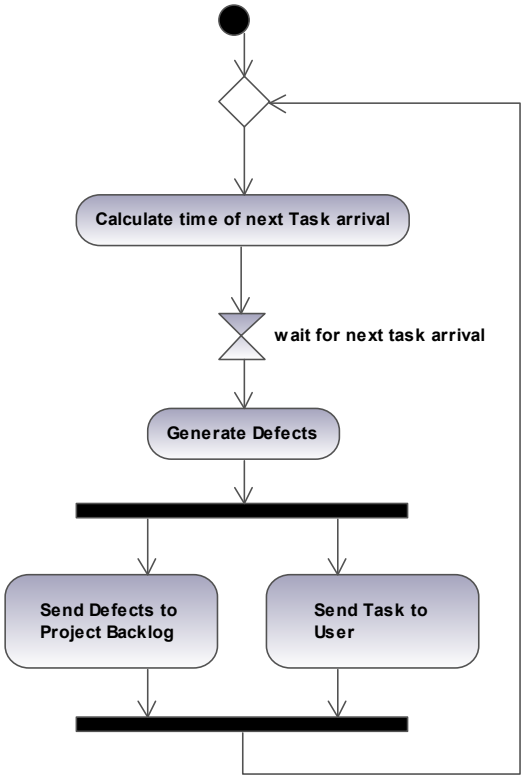

Figure 30. Defect generation activity diagram.

into that release's defect tracking system. Of the remaining 35% there is an 80% chance that the current major release's testing phase will catch the defect, in which case the defect will be placed in the major release's defect tracking system. In any case the number of defects not found in any release will be logged accordingly, and those remaining defects not tracked will exit the system.

Technical Debt Generation. For any task that is completed there is the possiblity of it causing an increase in technical debt. There are four types of technical debt a task can become: *Strategic, Tactical, Incremental,* or *Inadvertant*. Initially the generation process awaits the next task arrival (see Figure 32). Once a task arrives it is processed to determine if technical debt will be generated (the generation is shown in Figure 33). In

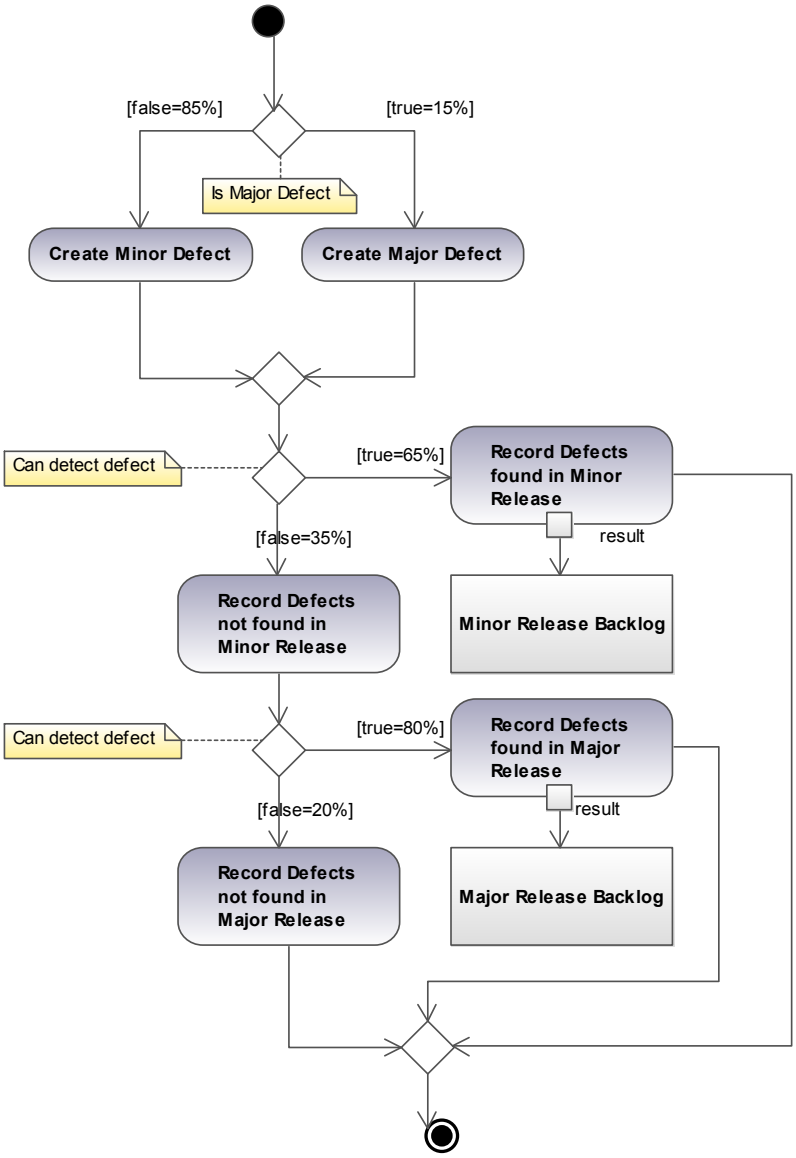

Figure 31. Defect creation and logging activity diagram.

the case that the task is already a refactoring, no technical debt is generated. If the task was required to be completed quicker than it should have, this will incur Tactical debt. Otherwise, either Incremental or Inadvertant technical debt may be created. In all cases of technical debt creation, *potential technical debt* (PTD) is generated. Potential technical debt are debt items which are technical debt but which may not be considered important

or necessary to be handled from the perspective of project planning. Once the PTD items is generated, it is evaluated to determine if it is *effective technical debt*; which is debt that should be repayed and which affects the current state of the system.

Once the status of potential vs effective technical debt has been established, the developer productivity effects due to technical debt are updated. At this point we determine if the technical debt can be detected. If the type of debt is strategic or tactical then the development teams will be aware of it and it is immediately placed into the technical debt list. On the other hand, if the debt is either incremental or inadvertent, it will need to pass through detection techniques. Here, if autodetection (tools for detecting

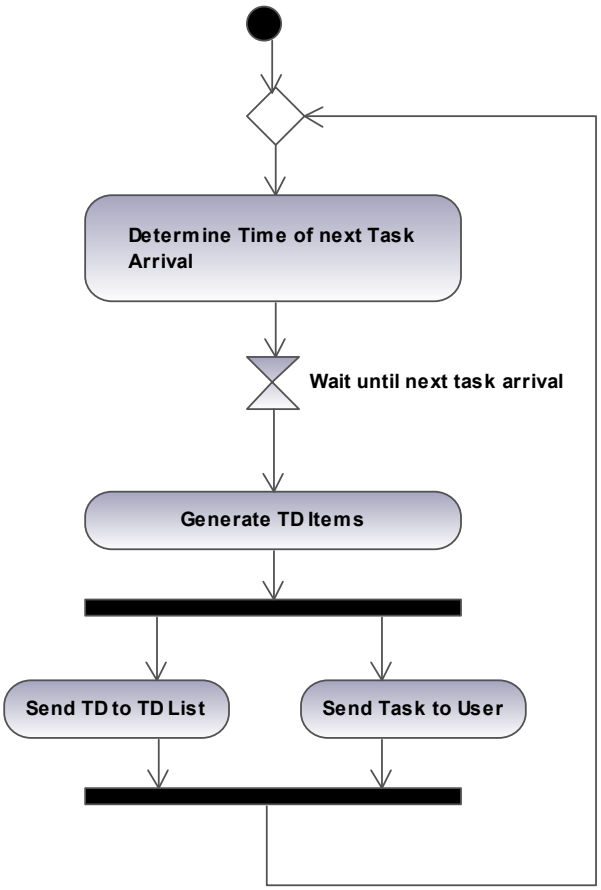

Figure 32. Technical debt generation from tasks process activity diagram

technical debt items such as code smells or antipatterns that are connected to either a continuous integration system, build scripts, or repository management systems) is enabled then the likelihood that the technical debt will be detected is increased. On the other hand, if manual checks (such as code reviews) are the only method in place then

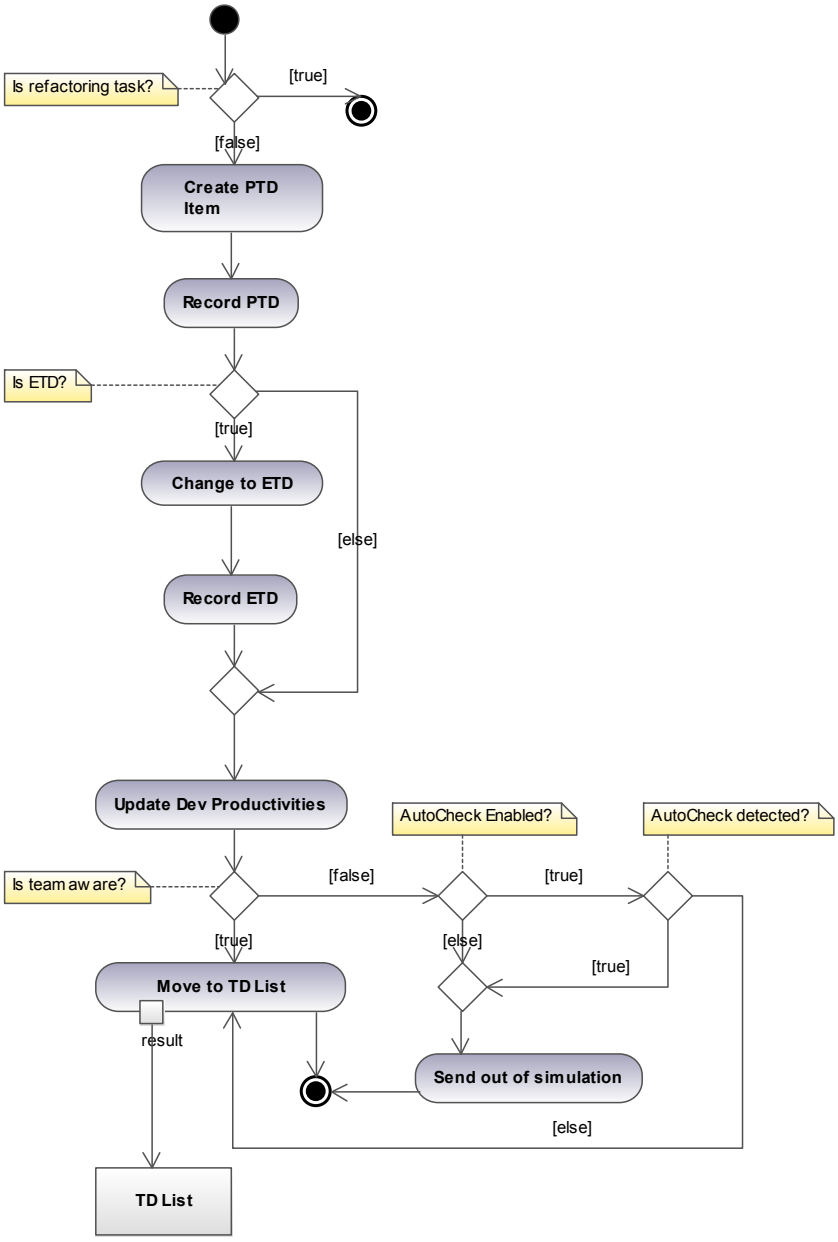

Figure 33. Technical debt generation activity.

detection will be greatly decreased. In either case, if items are detected they are placed into the technical debt list immediately, otherwise they exit the simulation. Once the debt items are generated and either have exited the system or entered the technical debt list, the task updates its associated work item and exits the simulation.

Software Engineer Generation and Lifecycle. In the simulation the resources of concern are software engineers. Initially software engineers are generated based on data which represents the teams associated with a given system. At model initialization each

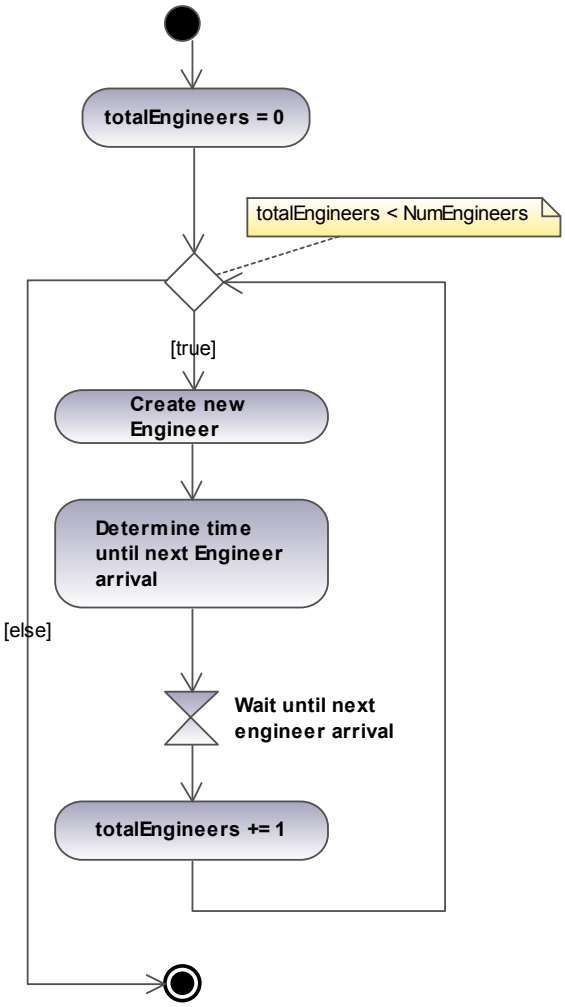

Figure 34. Software engineer generation activity diagram.

software engineer is generated by reading this information from the database or meta-information files at a constant rate, until the total number of engineers to be created is complete. This process is depicted in Figure 34.

Once all of the software engineers have been created and the simulation is properly initialized, the software engineers are activated and their lifecycle is activated. The software engineer lifecycle is depicted in Figure 35. Once the lifecyle has been activated, the engineer places itself into the Developer Pool to await assignment to a task. At this point the engineer waits until one of the following events occurs: *Made*

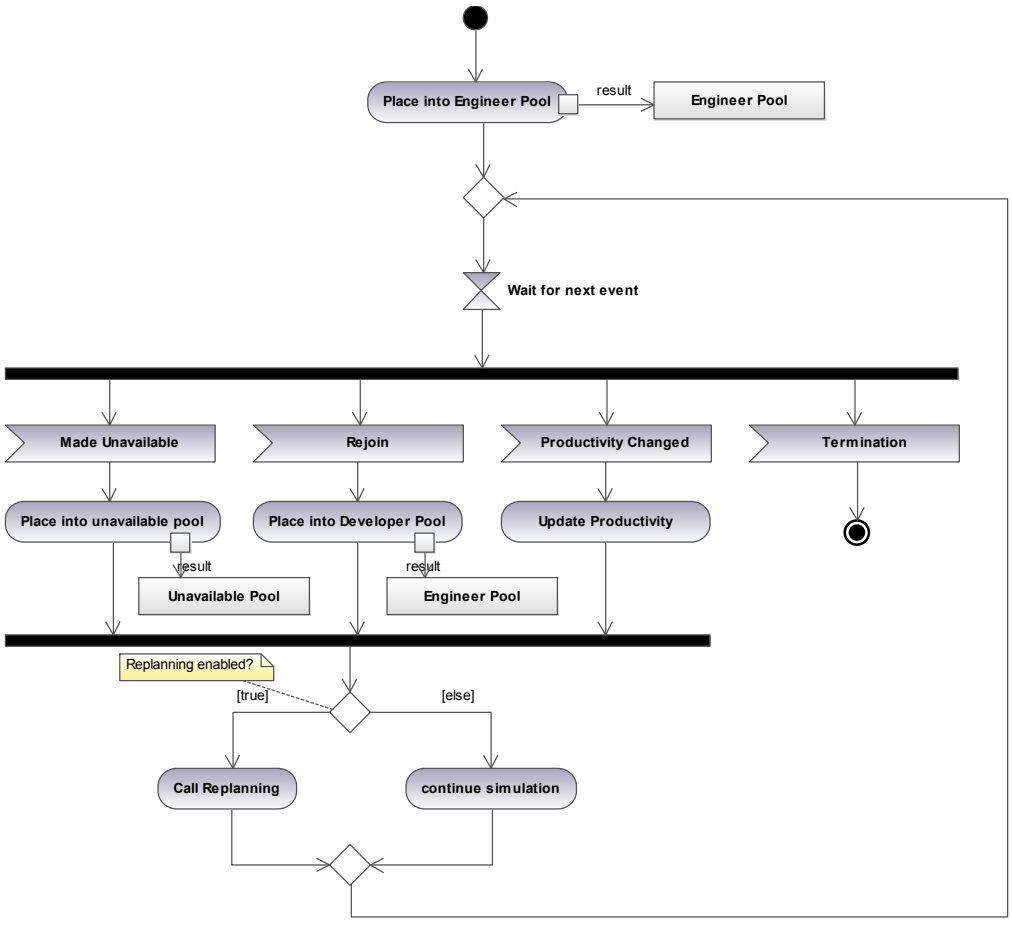

Figure 35. Engineer lifecycle activity diagram.

Unavailable, the software engineer becomes unavailable and removes itself from the Developer Pool; *Rejoin*, the engineer is made available again and rejoins the Developer Pool to begin working on tasks again; *Productivity Changed*, the developer's productivity level for a task type changes (such as when the system technical debt greatly increases); or *Termination*, which indicates that the simulation has ended.

Release Planning Process. The release planning process is used to select work items from the current major release evolution sequence and distribute it across the minor releases associated with the major release. To perform this allocation the selected SRP partitioning algorithm is used. While there are remaining minor releases to be completed, the proceses continues. First the process calculates the time required for the next minor release. Once the time is calculated the minor release is activated. The major release then waits until the minor release completes, at which time it begins the transfer of incomplete items to the next minor release. The process continues until all minor releases are completed. Once all minor releases are complete, any incomplete work items are transferred to the next major release evolution sequence. This process is described in Figure 36.

Project Process. The project process describes the lifecyle of a project within the simulation and is depicted in Figure 37. The project process begins by generating work items. Once all work items have been generated the intial SRP planning process begins by using the selected SRP planning algorithm. This planning process distributes work items across all major releases under consideration. While there are remaining major releases the process continues as follows. For the next major release, the execution time

required is calculated. The next major release is activated and the project process awaits until its execution is complete. Once the current major release is completed the major release velocity is calculated and recorded. When all major releases are complete, the project velocity is completed and the project is terminated.

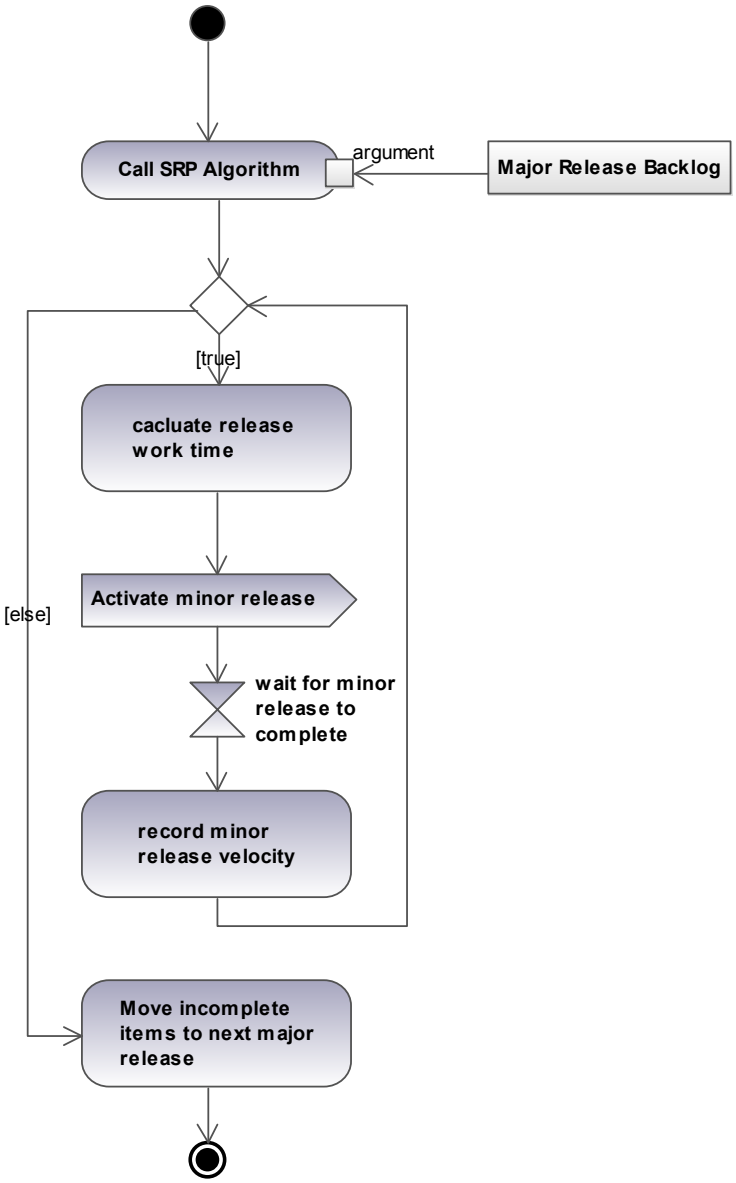

Figure 36. Major release process activity diagram.

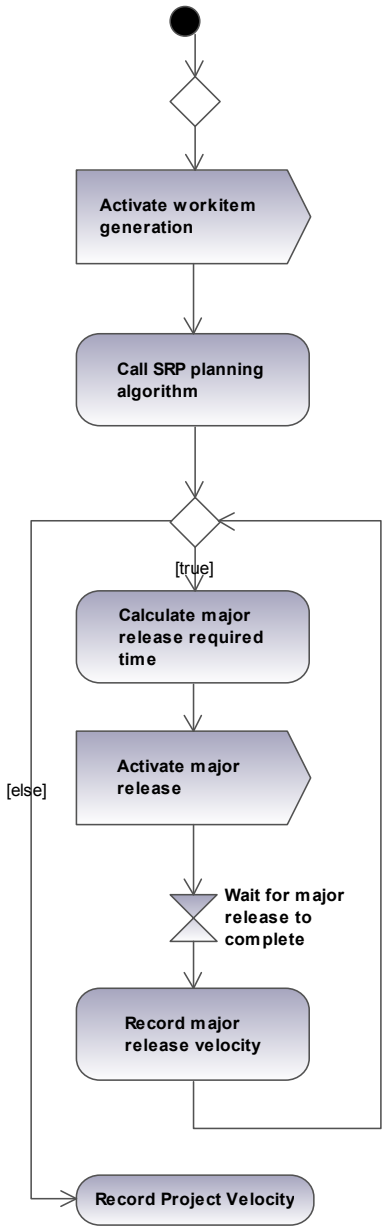

Figure 37. Project lifecycle activity diagram.

Major Release Process. The major release process describes the lifecyle of each major release in the project and is depicted in Figure 36. This process begins by selecting work items from the current major release evolution sequence and distribute it across the minor releases associated with the major release. To perform this allocation the selected

SRP partitioning algorithm is used. While there are remaining minor releases to be completed, the proceses continues. First the process calculates the time required for the next minor release. Once the time is calculated the minor release is activated. The major release then waits until the minor release completes, at which time it begins the transfer of imcomplete items to the next minor release. The process continues until all minor releases are completed. Once all minor releases are complete, any imcomplete work items are transferred to the next major release evolution sequence.

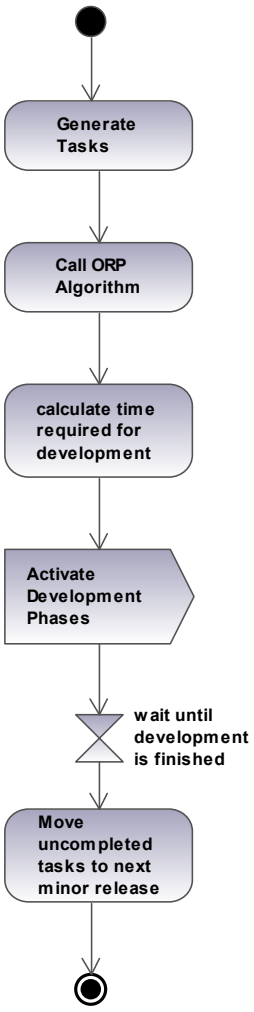

Figure 38. Minor release process activity diagram.

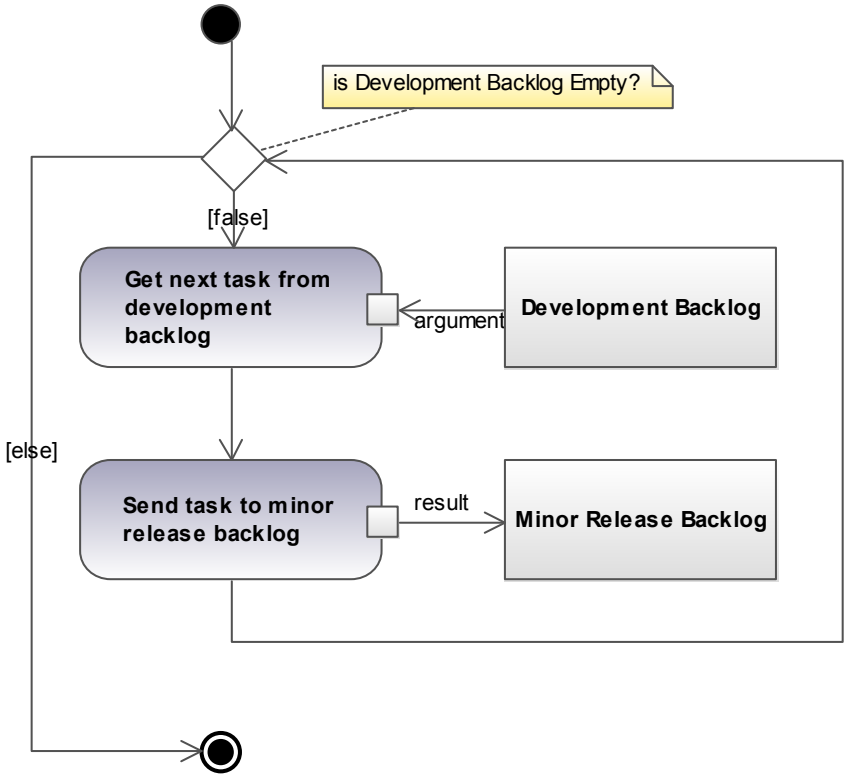

Figure 39. Move items to next minor release process activity diagram.

Minor Release Process. The minor release process describes the lifecyle of each minor release of a major release and is depicted in Figure 38. The minor release begins by generating the tasks associated with the work items in its evolution sequence. Once these tasks have been generated they are assigned to engineers from the engineer pool using the selected ORP algorithm. Once the assignments have been completed the execution time associated with the development phase is calculated. The development phase is then activated and the minor release waits until it is completed. Once the development phase is completed any incomplete items are moved to the next minor release evolution sequence (see Figure 39). If this is the last minor release of the current

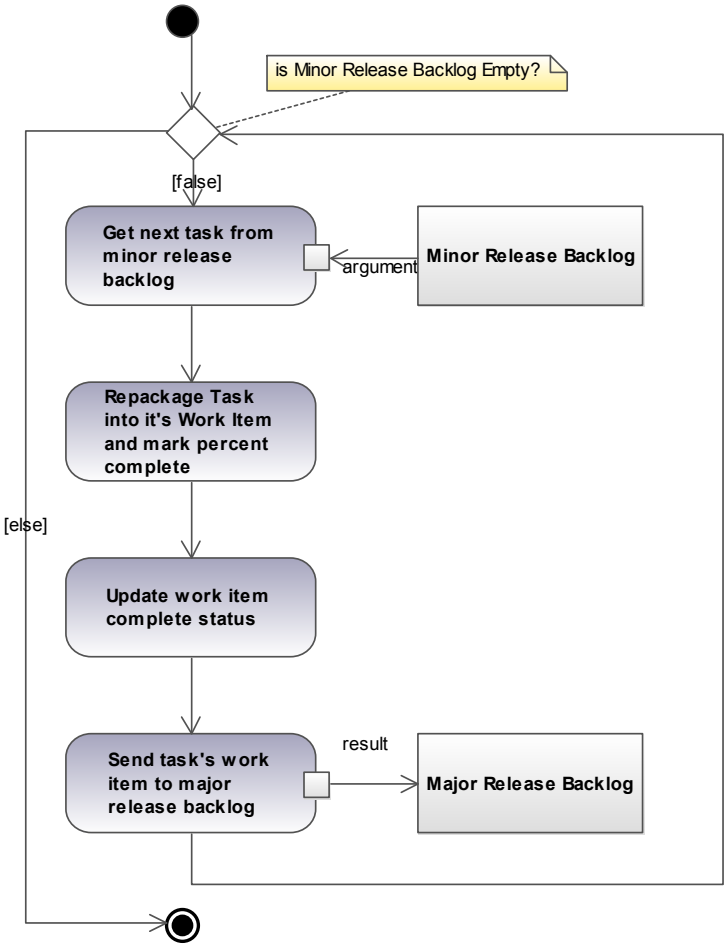

Figure 40. Move items to next major release process activity diagram.

major release then the work items of incomplete tasks are moved to the next major release's evolution sequence (see Figure 40).

Development Phases. The Development Phases is the simulation process which represents development of tasks and work items, are depicted in Figure 41. This process begins by selecting the first task/engineer pairs from the minor release plan. This selection both removes the engineer from the engineer pool as well as removing the task

from the development queue. Each task has an estimated time to completion, but the actual time to complete the task is caculated using the following formula:

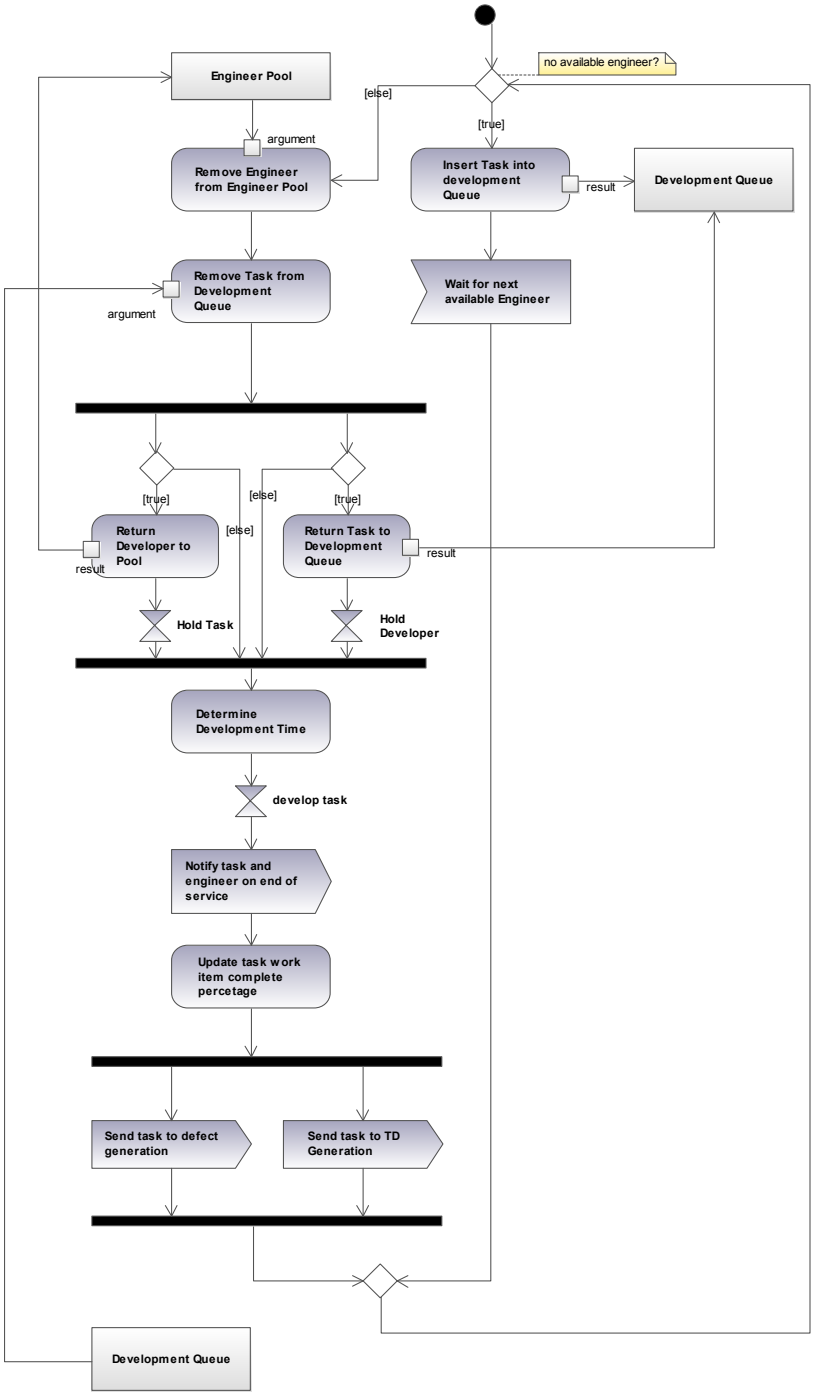

Figure 41. Development phase activity diagram.

$$ExecTime = Effort_a * EngProd_t * EngPref_w$$

$$Effort_a = Effort_e + \left(Effort_e * Rand(0.0, 0.6) * BIND(0.8, 1.0)\right)$$

Where $ExecTime$ is the actual execution time for the completion of the task by the assigned software engineer, $EngProd_t$ is the productivity of the assigned engineer for the task type of the assigned task, $Effort_a$ is the actual effort required for the task (for an engineer with a productivity for that task type of 1.0), $Effort_e$ is the estimated effort for the task (for an engineer with a productivity for that task type of 1.0), $UNI(1.0, 1.6)$ is a uniform random distribution between 0.0 and 0.6 to accommodate underestimation and BIND(0.8,1.0) is a binary distribution with a chance of 80% chance of 1 and 20% chance of 0 to indicate the probability of an underestimation. Once the execution time has been calculated the development process causes the engineer/task pair to wait until that time is complete. If either the development queue is empty or the release date is met, the development phase is completed. When development phases are complete any remaing work in the development queue is returned to the containing minor release evolution sequence. The remaining items in a minor release evolution sequence will be transferred to the next minor release or next major release if this was the last minor release for the current major release.

Re-Planning Process. The final process is the release re-planning process, which is depicted in Figure 42. This process, once activated during simulation initialization, begins waiting for the next event to occur. The events that this process listens for include: changes in work item/task priorities, changes in the technical debt level, changes in effort estimates, changes in engineer productivities, changes in engineer preferences, changes in

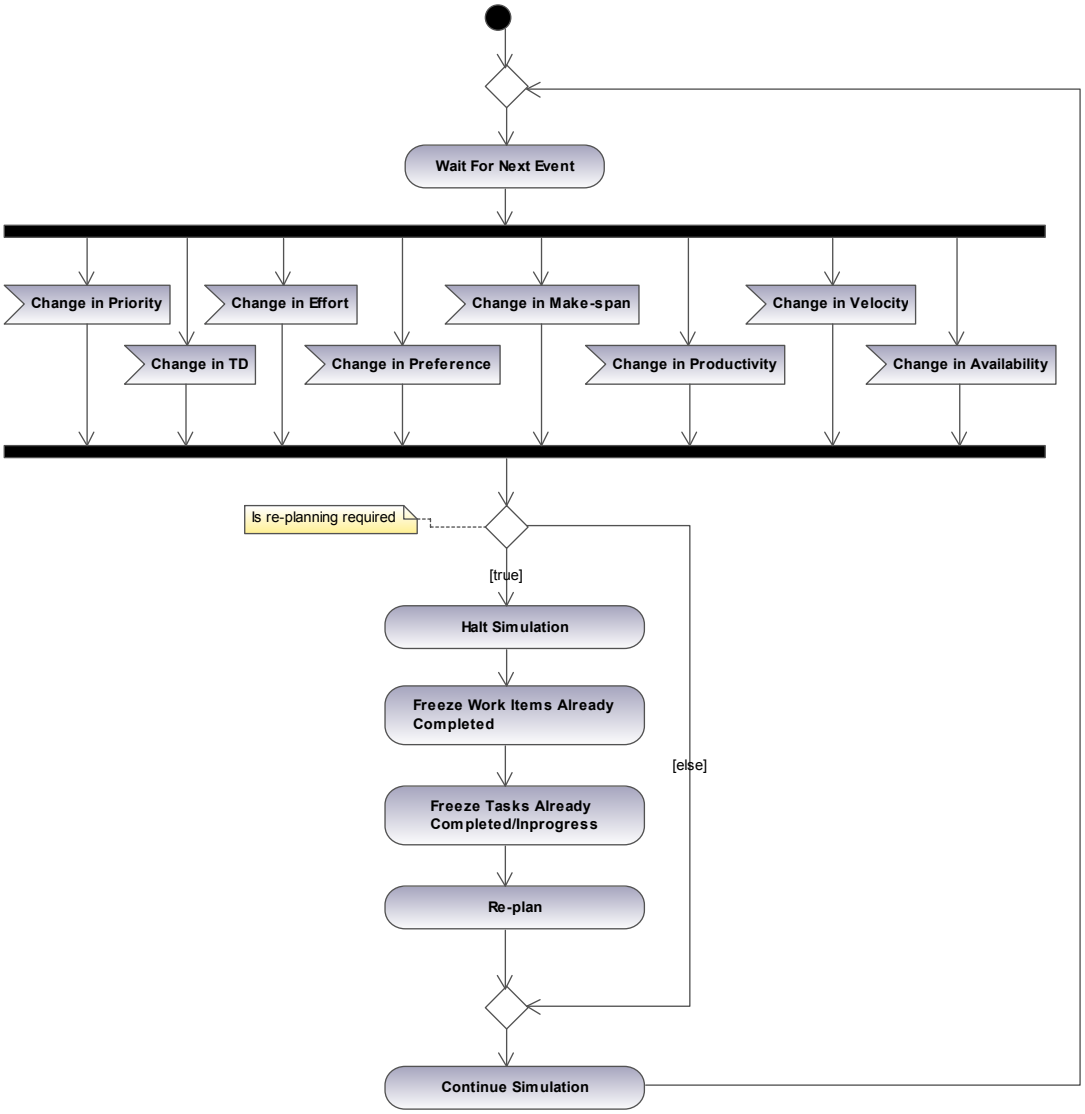

Figure 42. Re-planning process activity diagram.

release dates, changes in engineer or team velocity, or changes in engineer availability. Once an event has been received it is determined whether re-planning is required. If re-planning is required the simulation is halted and current tasks in progress and completed tasks/work items are held in their positions in both strategic and operational release plans. At this point depending on the amount of change that has occurred either operational

release planning for the current minor release is conducted, strategic release planning (and then further operational planning) at the major release level is conducted, or strategic release planning is conducted at the project level, major release level, and operational planning is conducted at the minor release level.

Table 11. Parameters for the extended simulation framework.

Parameter	Description
NumTeams	The number of teams assigned to the system.
Engineers	Meta-data describing each engineer including: team, name, and availability.
Productivities	The productivities for each task type for each engineer.
Preferences	The preferences for each work item type for each engineer.
Work Items	Meta-data describing work items including: Name, effort required, priority, type, etc.
Tasks	Meta-data describing tasks including: Name, work item, effort required, etc.
MajorReleaseDuration	Major release duration (in man-days) per major release (if using pre-defined release durations)
MinorReleaseDuration	Minor release duration (in man-days) per minor release (if using pre-defined release durations)
PredefinedDurations	Boolean value specifying whether pre-defined release durations are in use.
MaxMinorRelease	Maximum number of minor releases per major release.
MaxMajorRelease	Maximum number of major releases per project.
InitialTD	The level of technical debt currently in the system (man-days)
SystemSize	The current size of the system (in KLOC)
Repository	Meta-data describing the contents of the repository (at simulation initialization).
TDThreshold	Maximum level of technical debt before action will be taken.
TDOnlyTeams	Boolean value representing whether TD remediation teams are in use.
TDPercent	Maximum amount of effort to be devoted by development teams towards TD removal during a development phase.
TDProbDist	A probability distribution representing the occurrence of technical debt per task completed.
DefectProbDist	A probability distribution representing the occurrence of defects per task completed.

Simulation Parameters

There are several parameters for the simulation framework. These allow the user to select from different forms of questions or to conform to different software engineering processes. Each of the parameters are identified and described in Table 11.

Experimental Design

The Baseline Release Plan

Each experiment is designed as a proof-of-concept experiment to evaluate the effects of uncertainty or changes in external parameters on the generated release plans. In order to evaluate th is, we need to generate a baseline release plan for comparison. The experiments considered in this chapter are based on those conducted by Al-Emran et al. [42] [41]. Each experiment considers a group of 15 developers, 35 work items to be developed, and 3 tasks per work item. For the set of developers, the task productivities can be found in Table 12. For each work item to be developed Table 13 displays a listing of the baseline parameters for each work item and associated tasks.

Technical Debt Strategy Impact Analysis

In this experiment, we are looking at the impact of different strategies of technical debt management on technical debt removal. We are assuming that the base approach is based on automated detection of technical debt items, which are tracked using a technical debt list. We are attempting to evaluate the validity of using a team of software engineers dedicated to the removal of tech nical debt in conjunction with the main development team diverting a percentage of their time towards technical debt removal. In this

Table 12. Developer productivities for each task type.

Developer	Prod(k,1)	Prod(k,2)	Prod(k,3)
1	1	1.25	1
2	0	1	1.25
3	0.75	1	1
4	0.75	1.25	1
5	0.75	1	1.25
6	1	1.25	1.25
7	1	0.75	1
8	0	1	1
9	1.25	0.75	1
10	0.75	1.25	1
11	0.75	1	1.25
12	1	1.25	1.25
13	1	1.25	0
14	0.75	1	1.25
15	0	1.25	1

experiment we are evaluating how changes in these values affect technical debt removed and make-span of a release using stochastic analysis. The associated TRIANG distributions of the simulation parameters are listed in Table 13.

The parameters under consideration are *TD-Percent*, which is the percentage of effort the main development team will dedicate to technical debt removal, and *TD-Only*

Table 13. Triangular distributions of parameters to be used for stochastic analysis of technical debt remaining comparison to a given baseline plan.

Case	Variable	Min(%)	Peak(%)	Max(%)
Worst	TD-Percent	0	0	0
	TD-Only Team Size (% of dev team)	0	0	0
Poor	TD-Percent	0	0	10
	TD-Only Team Size (% of dev team)	50	60	70
Good	TD-Percent	0	5	15
	TD-Only Team Size (% of dev team)	70	80	100
Best	TD-Percent	15	20	30
	TD-Only Team	100	125	150

Table 14. Distribution of values for uncertainty factors across pessimism level [42].

Pessimism Level	Uncertainty Factor	Min(%)	Peak(%)	Max(%)
Bad	Effort	-20	0	30
	Productivity	-30	0	20
	Feature	0	0	30
	Developer	0	0	30
Worse	Effort	-10	0	40
	Productivity	-40	0	10
	Feature	0	15	30
	Developer	0	15	30
Worst	Effort	0	50	50
	Productivity	-50	-50	0
	Feature	0	30	30
	Developer	0	30	30

Team Size, which is a percentage of the size of the main development team. We will use multiple replications to evaluate the impact of these changes on the technical debt removed and make-span via comparison to the baseline plan.

Uncertainty Impact Analysis

In this experiment, we are looking to determine the effects of uncertainty on the release make-span and technical debt removed, in comparison to the baseline release. The method of evaluation is through stochastic analysis. The analysis is conducted using triangular distributions at various pessimism levels (see Table 14) for effort estimate, productivity estimates, feature percentage changes, and percent developer unavailability. The goal is to evaluate each of these factors independently, as well as, in the following combinations [42]:

- $\Delta Workitems, \Delta Effort$
- $\Delta Workitems, \Delta Engineers$

- $\Delta Workitems, \Delta Productivities$
- $\Delta Effort, \Delta Engineers$
- $\Delta Effort, \Delta Productivities$
- $\Delta Engineers, \Delta Productivities$
- $\Delta Workitems, \Delta Efforts, \Delta Engineers$
- $\Delta Workitems, \Delta Efforts, \Delta Productivities$
- $\Delta WorkItems, \Delta Engineers, \Delta Productivities$
- $\Delta Efforts, \Delta Engineers, \Delta Productivities$
- $\Delta WorkItems, \Delta Efforts, \Delta Engineers, \Delta Productivities$

Conclusion

This chapter details current work on the development of an extended framework for decision analysis in the areas of release planning and technical debt management. We have provided a conceptual model which is based on an existing simulation (see Chapter 6) and an underlying meta-model (see Chapter 4). The simulation model can be combined with existing approaches to strategic and operational release planning, release re-planning, and can model existing strategies for technical debt management (see Chapter 6). We have also included experiments to evaluate the sensitivity of the simulation and approaches underlying the simulation to various changes in the development process. The method of sensitivity analysis is based on one used by Al-Emran et al. [39] [42] [43] [81]. Using this method managers could easily update the simulation as development

approaches and given information (such as engineer availability changes) can be used to adjust the simulation to see the effects they will have on the current project.

CONCLUSIONS AND FUTURE WORK

In this thesis we have focused on the use of simulation as the underlying method used to provide decision support in the areas of release planning and technical debt management. We have provided an initial framework for software engineering decision support in the areas of technical debt management and release planning (see Chapter 3). Working towards this framework we have developed a domain meta-model which captures and unifies the important concepts from both release planning and technical debt management (see Chapter 4). Using the meta-model we developed a simulation framework to support the decision support framework (see Chapter 6 and 7). We intially used the simulation model to evaluate current technical debt management strategies used in industry. Finally, our recent work (as detailed in Chapter 7) has focused on extending this simulation model with the unified domain model to incorporate release planning methods, and we presented this as the foundation of a decision support system which can be used for both release planning and technical debt management.

In the future we plan to extend this work in the following ways. First, we would like to validate this work on industry projects. In order to do this we need to operationalize the framework and develop a tool which can acquire the necessary data. This would include the ability to connect to existing development support tools such as SonarQube (for technical debt evaluation), source code repositories (i.e., SVN, Git, and Mercurial), defect tracking systems such as Jira or FogBugz. Thus, the SEDS framework proposed herein moves out of the realm of simply planning releases and identifying when replanning should occur, but can then enter the realm of real time tracking of progress.

This will allow for a new form of decision support, one that will allow replanning events based on thresholds of deviation from the ideal plan. This type of information will be necessary in order to ensure that decisions regarding technical debt acquition are as accurate as possible.

REFERENCES CITED

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, Refactoring: Improving the Design of Existing Programs, Addison-Weseley, 1999.

[2] W. Cunningham, "The WyCash portfolio management system," *SIGPLAN OOPS Mess.,* vol. 4, no. 2, pp. 29-30, December 1992.

[3] J. Kerievsky, Refactoring to patterns, Pearson Deutschland GmbH, 2005.

[4] C. Neill and P. Laplante, "Paying down design debt with strategic refactoring," *Computer,* vol. 39, no. 12, pp. 131-134, 2006.

[5] P. Kruchten, R. L. Nord and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *Software, IEEE,* vol. 29, no. 6, pp. 18-21, December 2012.

[6] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman and F. Shull, "Comparing Four Approaches for Technical Debt Identification," *Software Quality Journal,* pp. 1-24, 2012.

[7] N. Zazworka, M. A. Shaw, F. Shull and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, Waikiki, Honolulu, HI, USA, 2011.

[8] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," *Advances in Computers,* vol. 82, pp. 25-46, 2011.

[9] E. Tom, A. Aurum and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software,* vol. 86, no. 6, pp. 1498-1516, 2013.

[10] A. Nugroho, J. Visser and T. Kuipers, "An empirical model of technical debt and interest," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, Waikiki, Honolulu, HI, USA, 2011.

[11] T. Klinger, P. Tarr, P. Wagstrom and C. Williams, "An enterprise perspective on technical debt," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, Waikiki, Honolulu, HI, USA, 2011.

[12] T. Theodoropoulos, M. Hofberg and D. Kern, "Technical debt from the stakeholder perspective," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, Waikiki, Honolulu, HI, USA, 2011.

[13] K. Schmid, "On the limits of the technical debt metaphor some guidance on going beyond," in *Managing Technical Debt (MTD 2013), 2013 4th International Workshop on*, 2013.

[14] K. Schmid, "A formal approach to technical debt decision making," in *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, New York, NY, USA, 2013.

[15] R. O. Spinola, A. Vetro, N. Zazworka, C. Seaman and F. Shull, "Investigating technical debt folklore: Shedding some light on technical debt opinion," in *Managing Technical Debt (MTD 2013), 2013 4th International Workshop on*, 2013.

[16] Z. Codabux and B. Williams, "Managing technical debt: An industrial case study," in *Managing Technical Debt (MTD 2013), 2013 4th International Workshop on*, 2013.

[17] S. McConnell, "Managing Technical Debt," 2008.

[18] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, Santa Fe, New Mexico, USA, 2010.

[19] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. da Silva, A. L. M. Santos and C. Siebra, "Tracking technical debt -- An exploratory case study," *Software Maintenance (ICSM 2011), 2011 27th IEEE International Conference on,* pp. 528-531, 2011.

[20] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, Waikiki, Honolulu, HI, USA, 2011.

[21] F. Fontana, V. Ferme and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," in *Managing Technical Debt (MTD 2012), 2012 Third International Workshop on*, 2012.

[22] N. Zazworka, C. Seaman and F. Shull, "Prioritizing design debt investment opportunities," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, Waikiki, Honolulu, HI, USA, 2011.

[23] K. Schmid, "Technical Debt -- From Metaphor to Engineering Guidance: A Novel Approach based on Cost Estimation," 2013.

[24] M. G. Stochel, M. R. Wawrowski and M. Rabiej, "Value-Based Technical Debt Model and Its Application," in *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*, 2012.

[25] B. Boehm and L. G. Huang, "Value-based software engineering: A case study," *Computer,* vol. 36, no. 3, pp. 33-41, 2003.

[26] J. Holvitie and V. Leppanen, "DebtFlag: Technical debt management with a development environment integrated tool," in *Managing Technical Debt (MTD), 2013 4th International Workshop on*, 2013.

[27] D. Falessi, M. A. Shaw, F. Shull, K. Mullen and M. S. Keymind, "Practical considerations, challenges, and requirements of tool-support for managing technical debt," in *Managing Technical Debt (MTD), 2013 4th International Workshop on*, 2013.

[28] N. Ramasubbu and C. F. Kemerer, "Towards a model for optimizing technical debt in software products," in *Managing Technical Debt (MTD 2013), 2013 4th International Workshop on*, 2013.

[29] R. J. Eisenberg, "A threshold based approach to technical debt," *SIGSOFT Softw. Eng. Notes,* vol. 37, no. 2, pp. 1-6, April 2012.

[30] C. Izurieta, A. Vetro, N. Zazworka, Y. Cai, C. Seaman and F. Shull, "Organizing the technical debt landscape," in *Managing Technical Debt ({MTD}), 2012 Third International Workshop on*, 2012.

[31] J. Morgenthaler, M. Gridnev, R. Sauciuc and S. Bhansali, "Searching for build debt: Experiences managing technical debt at Google," in *Managing Technical Debt (MTD 2012), 2012 Third International Workshop on*, 2012.

[32] R. Marinescu, "Assessing and Improving Object-Oriented Design," 2012.

[33] R. Nord, I. Ozkaya, P. Kruchten and M. Gonzalez-Rojas, "In Search of a Metric for Managing Architectural Technical Debt," in *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, 2012.

[34] J. L. Letouzey, "The SQALE method for evaluating Technical Debt," in *Managing Technical Debt (MTD 2012), 2012 Third International Workshop on*, 2012.

[35] B. Curtis, J. Sappidi and A. Szynkarski, "Estimating the Principal of an Application's Technical Debt," *Software, IEEE,* vol. 29, no. 6, pp. 34-42, December 2012.

[36] B. Curtis, J. Sappidi and A. Szynkarski, "Estimating the size, cost, and types of Technical Debt," in *Managing Technical Debt (MTD 2012), 2012 Third International Workshop on*, 2012.

[37] C. Izurieta, I. Griffith, D. Reimanis and R. Luhr, "On the Uncertainty of Technical Debt Measurements," in *Information Science and Applications (ICISA), 2013 International Conference on*, 2013.

[38] G. Ruhe, "Software release planning," *Handbook of software engineering and knowledge engineering,* vol. 3, pp. 365-394, 2005.

[39] A. Al-Emran, K. Khosrovian, D. Pfahl and G. Ruhe, "Simulation-based uncertainty analysis for planning parameters in operational product management," in *Proceedings of the 10th International Conference on Integrated Design and Process Technology*, Antalya, Turkey, 2007.

[40] A. Al-Emran and D. Pfahl, "Operational Planning, Re-planning and Risk Analysis for Software Releases," in *Product-Focused Software Process Improvement*, vol. 4589, J. Münch and P. Abrahamsson, Eds., Springer Berlin Heidelberg, 2007, pp. 315-329.

[41] A. Al-Emran, A. Jadallah, E. Paikari, D. Pfahl and G. Ruhe, "Application of re-estimation in re-planning of software product releases," in *New Modeling Concepts for Today's Software Processes*, Springer, 2010, pp. 260-272.

[42] A. Al-Emran, P. Kapur, D. Pfahl and G. Ruhe, "Simulating worst case scenarios and analyzing their combined effect in operational release planning," in *Making Globally Distributed Software Development a Success Story*, Springer, 2008, pp. 269-281.

[43] A. Al-Emran, P. Kapur, D. Pfahl and G. Ruhe, "Studying the impact of uncertainty in operational release planning -- An integrated method and its initial evaluation," *Information and Software Technology,* vol. 52, no. 4, pp. 446-461, 2010.

[44] A. Al-Emran, D. Pfahl and G. Ruhe, "DynaReP: A Discrete Event Simulation Model for Re-planning of Software Releases," in *Software Process Dynamics and*

Agility, vol. 4470, Q. Wang, D. Pfahl and D. M. Raffo, Eds., Springer Berlin Heidelberg, 2007, pp. 246-258.

[45] O. Saliu and G. Ruhe, "Software release planning for evolving systems," *Innovations in Systems and Software Engineering,* vol. 1, no. 2, pp. 189-204, 2005.

[46] C.-W. Chiang and Y.-Q. Huang, "Comparison of ant-inspired search techniques for software release planning," in *Fuzzy Theory and it's Applications (iFUZZY 2012), 2012 International Conference on*, 2012.

[47] G. Ruhe, "Software engineering decision support: methodology and applications," *Innovations in decision support systems,* vol. 3, pp. 143-174, 2003.

[48] Amandeep, G. Ruhe and M. Stanford, "Intelligent Support for Software Release Planning," in *Product Focused Software Process Improvement*, vol. 3009, F. Bomarius and H. Iida, Eds., Springer Berlin Heidelberg, 2004, pp. 248-262.

[49] P. Husbands, "Genetic algorithms for scheduling," *AISB Quarterly,* vol. 89, pp. 38-45, 1994.

[50] G. Ruhe and A. N. The, "Hybrid intelligence in software release planning," *International Journal of Hybrid Intelligent Systems,* vol. 1, no. 1, pp. 99-110, 2004.

[51] J. T. Souza, C. L. B. Maia, T. N. Ferreira, R. A. F. Carmo and M. M. A. Brasil, "An Ant Colony Optimization Approach to the Software Release Planning with Dependent Requirements," in *Search Based Software Engineering*, vol. 6956, M. B. Cohen and M. Ó Cinnéide, Eds., Springer Berlin Heidelberg, 2011, pp. 142-157.

[52] G. Ruhe and M. O. Saliu, "The Science and Practice of Software Release Planning," *University of Calgary,* 2005.

[53] G. Ruhe and J. Momoh, "Strategic Release Planning and Evaluation of Operational Feasibility," in *System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on*, 2005.

[54] K. Schwaber and M. Beedle, Agile Software Development with Scrum, 1 ed., Upper Saddle River, New Jersey: Prentice Hall, 2001.

[55] A. Bagnall, V. Rayward-Smith and I. Whittley, "The next release problem," *Information and Software Technology,* vol. 43, no. 14, pp. 883-890, 2001.

[56] H.-W. Jung, "Optimizing value and cost in requirements analysis," *Software, IEEE,* vol. 15, no. 4, pp. 74-78, 1998.

[57] A. Ngo-The and G. Ruhe, "A systematic approach for solving the wicked problem of software release planning," *Soft Computing,* vol. 12, no. 1, pp. 95-108, 2008.

[58] F. Colares, J. Souza, R. Carmo, C. Padua and G. Mateus, "A New Approach to the Software Release Planning," in *Software Engineering, 2009. SBES '09. XXIII Brazilian Symposium on*, 2009.

[59] G. Ruhe, Product Release Planning: Methods, Tools and Applications, Auerbach Publications, 2011.

[60] J. Duggan, J. Byrne and G. Lyons, "A task allocation optimizer for software construction," *Software, IEEE,* vol. 21, no. 3, pp. 76-82, June 2004.

[61] D. Greer and G. Ruhe, "Software release planning: an evolutionary and iterative approach," *Information and Software Technology,* vol. 46, no. 4, pp. 243-253, 2004.

[62] M. Ramzan, M. Iqbal, M. Jaffar, A. Rauf, S. Anwar and A. Shahid, "Project Scheduling Conflict Identification and Resolution Using Genetic Algorithms," in *Information Science and Applications (ICISA), 2010 International Conference on*, 2010.

[63] H. W. J. Rittel and M. M. Webber, "Planning problems are wicked problems," *Developments in design methodology,* pp. 135-144, 1984.

[64] M. Svahnberg, T. Gorschek, R. Feldt, R. Torkar, S. B. Saleem and M. U. Shafique, "A systematic review on strategic release planning models," *Information and software technology,* vol. 52, no. 3, pp. 237-248, 2010.

[65] Handbook on Scheduling, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

[66] R. K. Wysocki, Effective software project management, Indianapolis, IN: Wiley, 2006.

[67] C. K. Chang, M. J. Christensen and T. Zhang, "Genetic Algorithms for Project Management," *Annals of Software Engineering,* vol. 11, no. 1, pp. 107-139, 2001.

[68] T. Abdel-Hamid, "The dynamics of software project staffing: a system dynamics based simulation approach," *IEEE Transactions on Software Engineering,* vol. 15, no. 2, pp. 109-119, Feb 1989.

[69] N. Fenton, W. Marsh, M. Neil, P. Cates, S. Forey and M. Tailor, "Making resource decisions for software projects," 2004.

[70] G. Antoniol, M. Di Penta and M. Harman, "Search-based techniques applied to optimization of project planning for a massive maintenance project," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, 2005.

[71] A. Barreto, M. d. O. Barros and C. M. Werner, "Staffing a software project: A constraint satisfaction and optimization-based approach," *Computers & Operations Research,* vol. 35, no. 10, pp. 3073-3089, Oct 2008.

[72] S. Hartmann, "A competitive genetic algorithm for resource-constrained project scheduling," *Naval Research Logistics,* vol. 45, no. 7, pp. 733-750, Oct 1998.

[73] M. M. Rahman, G. Ruhe and T. Zimmermann, "Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects," 2009.

[74] M. Przepiora, R. Karimpour and G. Ruhe, "A hybrid release planning method and its empirical justification," 2012.

[75] M. Nayebi and G. Ruhe, "An open innovation approach in support of product release decisions," 2014.

[76] A. van Lamsweerde, "Requirements engineering in the year 00: a research perspective," 2000.

[77] G. Kotonya and M. Sommerville, Requirements engineering: processes and techniques, Chichester ; New York: J. Wiley, 1998.

[78] T. Albourae, G. Ruhe and M. Moussavi, "Lightweight Replanning of Software Product Releases," in *Software Product Management, 2006. IWSPM '06. International Workshop on*, 2006.

[79] H. Kellerer, U. Pferschy and D. Pisinger, Knapsack Problems, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[80] T. L. Saaty, "Analytic hierarchy process," in *Encyclopedia of Operations Research and Management Science*, Dordrecht, Kluwer Academic Publishers, pp. 19-28.

[81] A. Al-Emran, D. Pfahl and G. Ruhe, "A method for re-planning of software releases using discrete-event simulation," *Software Process: Improvement and Practice,* vol. 13, no. 1, pp. 19-33, Jan 2008.

[82] A. Jadallah, A. Al-Emran, M. Moussavi and G. Ruhe, "The how? when? and what? for the process of re-planning for product releases," in *Trustworthy Software Development Processes*, vol. 5543, Q. Wang, V. Garousi, R. Madachy, D. Pfahl, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi and G. Weikum, Eds., Berlin, Heidelberg, Springer Berlin Heidelberg, 2009.

[83] M. Golfarelli, S. Rizzi and E. Turricchia, "Multi-sprint planning and smooth replanning: An optimization model," *Journal of Systems and Software,* vol. 86, no. 9, pp. 2357-2370, Sep 2013.

[84] D. F. Bacon, D. C. Parkes, Y. Chen, M. Rao, I. Kash and M. Sridharan, "Predicting your own effort," in *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, Valencia, Spain, 2012.

[85] M. Klein, G. A. Moreno, D. C. Parkes and K. Wallnau, "Designing for incentives: better information sharing for better software engineering," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, Santa Fe, New Mexico, USA, 2010.

[86] M. Yilmaz and R. V. O'Connor, "Maximizing the value of the software development process by game theoretic analysis," in *Proceedings of the 11th International Conference on Product Focused Software*, Limerick, Ireland, 2010.

[87] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol and Y. Gueheneuc, "Playing with refactoring: Identifying extract class opportunities through game theory," in *Software Maintenance (ICSM 2010), 2010 IEEE International Conference on*, 2010.

[88] J. H. Dréze and J. Greenberg, "Hedonic Coalitions: Optimality and Stability," *Econometrica,* vol. 48, no. 4, pp. 987-1003, 1980.

[89] A. Bogomolnaia and M. O. Jackson, "The Stability of Hedonic Coalition Structures," *Games and Economic Behavior,* vol. 38, no. 2, pp. 201-230, 2002.

[90] W. Saad, Z. Han, T. Basar, M. Debbah and A. Hjorungnes, "A selfish approach to coalition formation among unmanned air vehicles in wireless networks," 2009.

[91] W. Saad, Z. Han, M. Debbah, A. Hjorungnes and T. Basar, "Coalitional game theory for communication networks," *IEEE Signal Processing Magazine,* vol. 26, no. 5, pp. 77-97, September 2009.

[92] H. Zhang, B. Kitchenham and D. Pfahl, "Software Process Simulation Modeling: Facts, Trends and Directions," in *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, 2008.

[93] M. I. Kellner, R. J. Madachy and D. M. Raffo, "Software process simulation modeling: why? what? how?," *Journal of Systems and Software,* vol. 46, no. 2, pp. 91-105, 1999.

[94] T. Magennis, Forecasting and Simulating Software Development Projects: Effective Modeling of Kanban & Scrum Projects using Monte-carlo simulation, CreateSpace Independent Publishing Platform, 2011.

[95] F. Glaiel, A. Moulton and S. Madnick, "Agile Project Dynamics: A System Dynamics Investigation of Agile Software Development Methods," 2013.

[96] B. Spasic and B. S. S. Onggo, "Agent-based simulation of the software development process: A case study at AVL," 2012.

[97] G. Ruhe, "Software Engineering Decision Support - A New Paradigm for Learning Software Organizations," in *Advances in Learning Software Organizations*, vol. 2640, S. Henninger, F. Maurer, G. Goos, J. Hartmanis and J. Leeuwen, Eds., Berlin, Heidelberg, Springer Berlin Heidelberg, 2003, pp. 104-113.

[98] O. Saliu and G. Ruhe, "Supporting Software Release Planning Decisions for Evolving Systems," in *Software Engineering Workshop, 2005. 29th Annual IEEE/NASA*, 2005.

[99] G. Ruhe and D. Greer, "Quantitative studies in software release planning under risk and resource constraints," in *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, 2003.

[100] G. Ruhe and M. Saliu, "The art and science of software release planning," *Software, IEEE,* vol. 22, no. 6, pp. 47-53, December 2005.

[101] P. Carlshamre, K. Sandahl, M. Lindvall, B. Regnell and J. Natt och Dag, "An industrial survey of requirements interdependencies in software product release planning," in *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, 2001.

[102] T. L. Saaty, "How to make a decision: The analytic hierarchy process," *European Journal of Operational Research,* vol. 48, no. 1, pp. 9-26, September 1990.

[103] L. Lehtola and M. Kauppinen, "Suitability of requirements prioritization methods for market-driven software product development," *Software Process: Improvement and Practice,* vol. 11, no. 1, pp. 7-19, 2006.

[104] L. Rajbhandari and E. Snekkenes, "An approach to measure effectiveness of control for risk analysis with game theory," in *Socio-Technical Aspects in Security and Trust (STAST), 2011 1st Workshop on*, 2011.

[105] L. Lehtola, M. Kauppinen and S. Kujala, "Requirements Prioritization Challenges in Practice," in *Product Focused Software Process Improvement*, vol. 3009, F. Bomarius and H. Iida, Eds., Springer Berlin Heidelberg, 2004, pp. 497-508.

[106] V. Heikkila, A. Jadallah, K. Rautiainen and G. Ruhe, "Rigorous Support for Flexible Planning of Product Releases - A Stakeholder-Centric Approach and Its Initial Evaluation," in *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, 2010.

[107] P. Carlshamre, "Release Planning in Market-Driven Software Product Development: Provoking an Understanding," *Requirements Engineering,* vol. 7, no. 3, pp. 139-151, 2002.

[108] S. McConnell, Software Estimation: Demystifying the Black Art, Microsoft Press, 2006.

[109] M. Denne and J. Cleland-Huang, "The incremental funding method: data-driven software development," *IEEE Software,* vol. 21, no. 3, pp. 39-47, May 2004.

[110] K. Moløkken-Østvold, N. C. Haugen and H. C. Benestad, "Using planning poker for combining expert estimates in software projects," *Journal of Systems and Software,* vol. 81, no. 12, pp. 2106-2117, 2008.

[111] M. Cohn, Agile estimating and planning, Prentice Hall, 2006.

[112] G. Rowe and G. Wright, "The Delphi technique as a forecasting tool: issues and analysis," *International Journal of Forecasting,* vol. 15, no. 4, pp. 353-375, 1999.

[113] T. Menzies, Z. Chen, J. Hihn and K. Lum, "Selecting Best Practices for Effort Estimation," *Software Engineering, IEEE Transactions on,* vol. 32, no. 11, pp. 883-895, 2006.

[114] R. Tamrakar and M. Jørgensen, "Does the use of Fibonacci numbers in Planning Poker affect effort estimates?," in *Evaluation & Assessment in Software Engineering, 16th International Conference on*, Cludad Real, Spain, 2012.

[115] M. Woodridge, An Introduction to MultiAgent Systems, John Wiley & Sons, Ltd., 2009.

[116] C. Sterling, Managing Software Debt: Building for Inevitable Change, Addison-Wesley Professional, 2010.

[117] R. B. Myerson, Game theory: analysis of conflict, Harvard university press, 2013.

[118] R. DeMillo, R. Lipton and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer,* vol. 11, no. 4, pp. 34-41, Apr 1978.

[119] C. Izurieta, G. Poole, R. Payn, I. Griffith, R. Nix, A. Helton, E. Bernhardt and A. Burgin, "Development and Application of a Simulation Environment (NEO) for Integrating Empirical and Computational Investigations of System-Level Complexity," in *Information Science and Applications (ICISA), 2012 International Conference on*, 2012.

[120] T. D. Cook and D. T. Campbell, Quasi-experimentation: design & analysis issues for field settings, Boston: Houghton Mifflin, 1979.

[121] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén, Experimentation in Software Engineering, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[122] H. Liu, Z. Ma, W. Shao and Z. Niu, "Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort," *Software Engineering, IEEE Transactions on,* vol. 38, no. 1, pp. 220-235, Feb 2012.

[123] Y. Luo, A. Hoss and D. Carver, "An ontological identification of relationships between anti-patterns and code smells," in *Aerospace Conference, 2010 IEEE*, 2010.

[124] W. Opdyke, "Refactoring object-oriented frameworks," 1992.

[125] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development,* vol. 56, no. 5, pp. 9:1-9:13, October 2012.

[126] S. McConnell, Code Complete: A Practical Handbook of Software Construction, 2 ed., Redmond, Washington: Microsoft Press, 2004.

[127] H. Sharp, A. Finkelstein and G. Galal, "Stakeholder identification in the requirements engineering process," in *Database and Expert Systems Applications, 1999. Proceedings. Tenth International Workshop on*, 1999.

[128] T. Ritchey, Wicked Problems--Social Messes, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

APPENDIX A

NORMAL Q-Q PLOTS FOR COALTION FORMATION GAME EXPERIMENTS

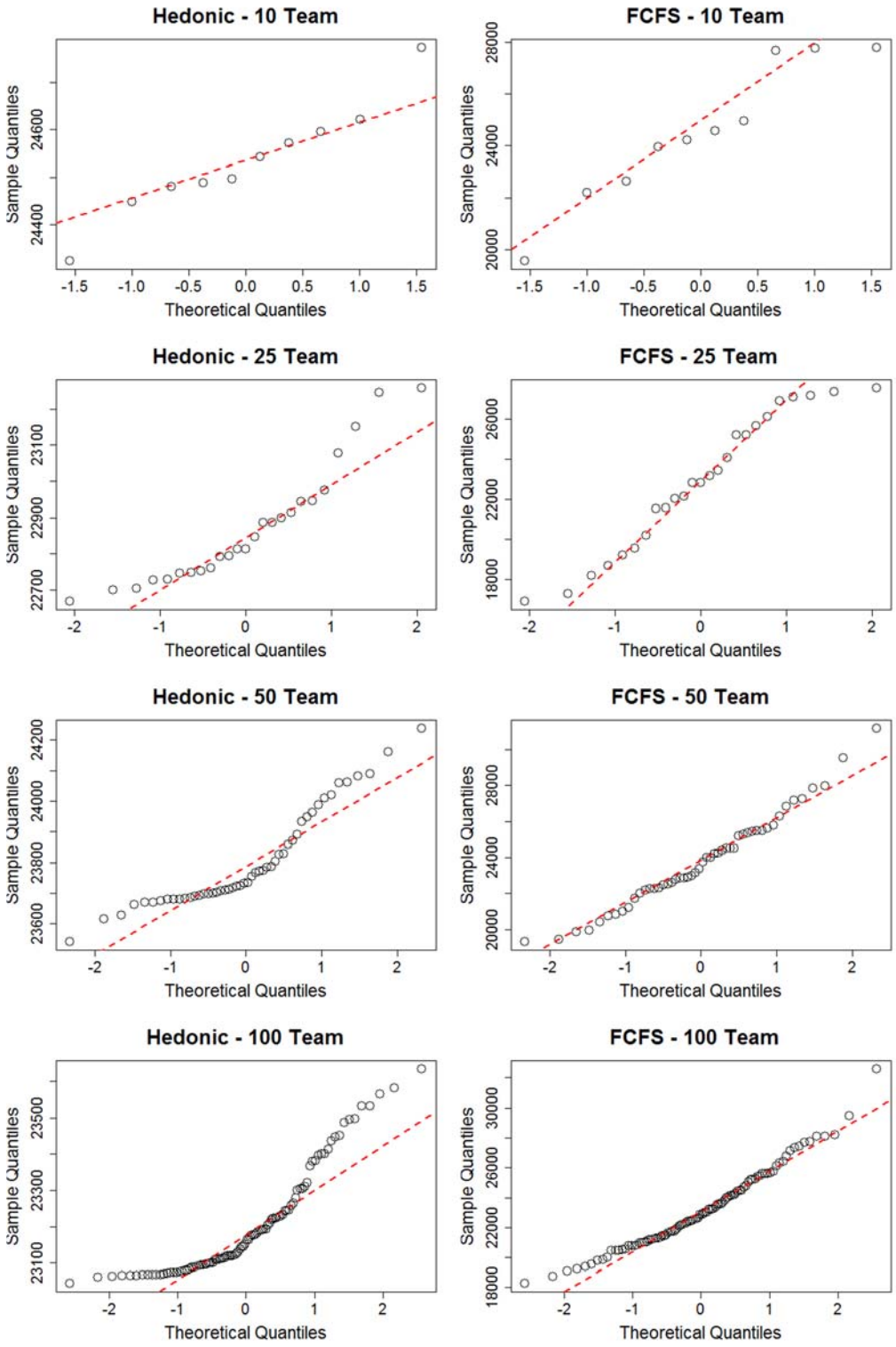

Figure 43. Normal Q-Q plots for experiment 1.

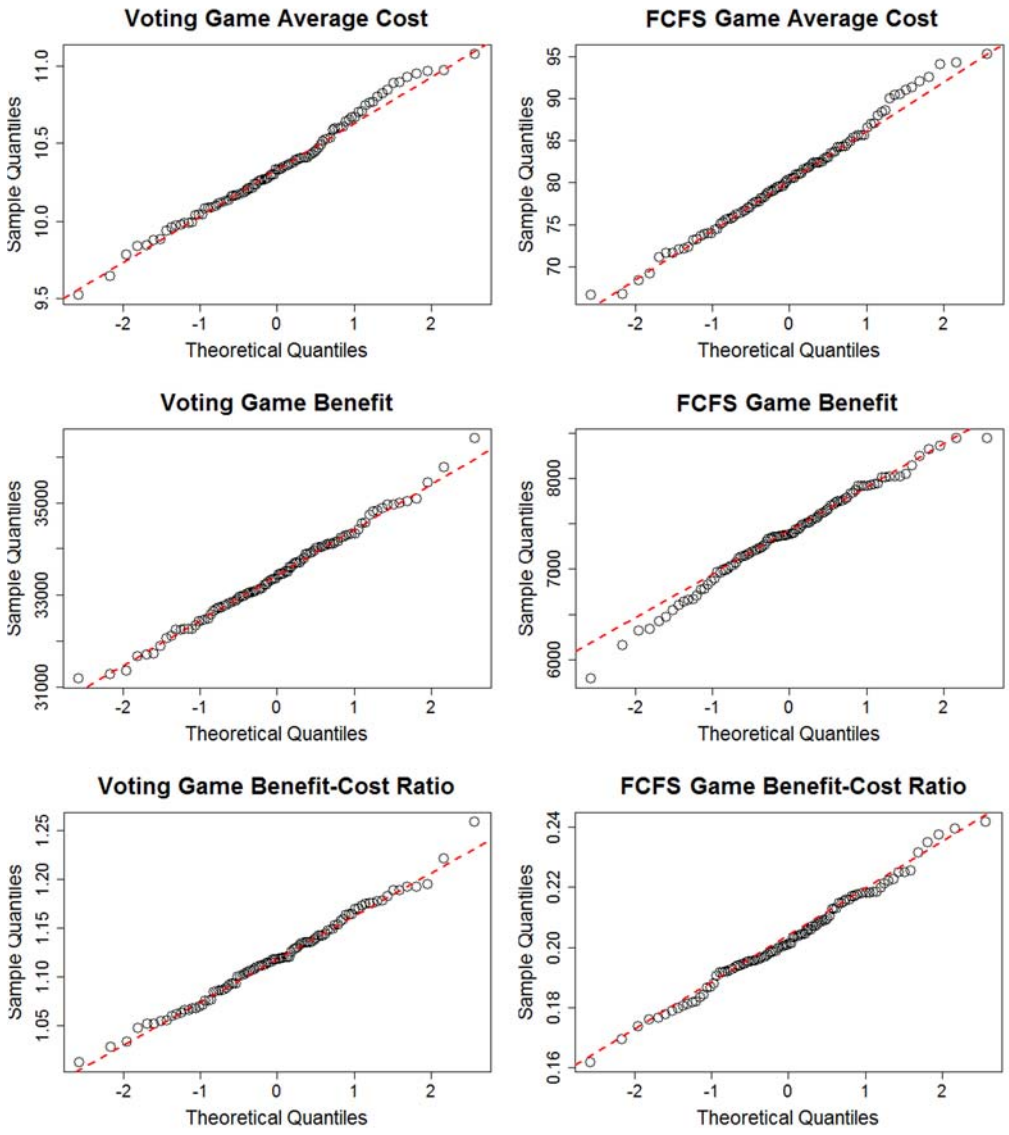

Figure 44. Normal Q-Q plots for experiment 2.