Proceedings of the 2013 Winter Simulation Conference
R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, eds.

Technical Debt Management – A Simulation Study of Practical Methods Proposed for Agile Software Development

Isaac Griffith and Clemente Izurieta

Department of Computer Science
Montana State University
Bozeman, MT 59717, USA

Hanane Taffahi and David Claudio

Department of Mechanical and Industrial Engineering
Montana State University
Bozeman, MT 59717, USA

ABSTRACT

Technical debt is a well understood yet understudied phenomena. A current issue is the verification of proposed methods for technical debt management in the context of agile development. In practice, such evaluations are either too costly or time consuming to be conducted using traditional empirical methods. In this paper, we describe a set of simulations based on models of the Scrum agile development process and the integration of technical debt management. It is the purpose of this study to not only identify which strategy is superior but to also provide evidence to support the existing claims. The models are based upon existing models concerning defects and technical debt generation from industry. The results of the simulation provide compelling support for current strategies for technical debt management identified in the literature that can be immediately applied by practitioners in industry.

1 INTRODUCTION

Technical debt embodies the dichotomy between making long-term decisions, affecting the quality of the software, versus short-term decisions, affecting the time-to-market value of the software. That is, while software should be delivered on time, any debt (sacrifice in quality) taken out to make that possible must eventually be repaid in order to ensure the overall health of the product. This has been a growing problem since as early as 1992 (Cunningham 1992), 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 (Ramasubbu and Kemerer 2013). This is due to the nature of the problem making empirical studies prohibitive. Thus, simulation provides an excellent method to evaluate these 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 then determine of those proposed technical debt management strategies is the most feasible to implement within an existing agile development process model. For this problem, we have selected the Scrum agile development process (Schwaber and Beedle 2001) to investigate the introduction of technical debt management strategies.

This paper is organized as follows: Section 2 describes background concepts and related work. Section 3 describes the conceptual model. Section 4 describes the experimental design and data collection methods to be used in this study. Section 5 describes the results and analysis of this simulation study. Section 6 describes the threats to the validity of this study, while section 6 concludes the paper and provides avenues of future work.

2 RELATED WORK

This research is centered around three major concepts: The first is software process simulation modeling, which is a branch of empirical software engineering focused on simulating different aspects of the software development life cycle. It is aimed at evaluating and assessing staffing requirements, predicting release dates, etc. (Kellner, Madachy, and Raffo 1999). The second is agile software development, specifically Scrum which is still one of the most widely used agile processes in the software industry. Finally, the main focus of this research is on technical debt and technical debt management. This section describes these concepts in more detail as well as relevant related work.

Simulation has been widely used as a means of prediction and analysis in the software industry. Kellner, Madachy and Raffo (Kellner, Madachy, and Raffo 1999) explored the area of Software Process Simulation Modeling (SPSM) in order to understand the methods used as well as the problems to which simulation has been applied. They also connected the use of simulation to that of 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 literture, they found that most simulation studies conducted are centered around the process or project level.

A further study by Zhang, Ketchenham, and Pfahl (2008) centered on the current trends in SPSM noted that of all the simulation modeling paradigms used, both discrete-event and continuous simulation formed the mainstream paradigms. They also note that there is a need to increase modeling and simulation at the process and entity level. A specific instance of process level simulation is the work of Magennis (Magennis 2011), which utilizes Monte-Carlo simulation to evaluate the effects of changes to agile development processes. Another example of agile process simulation is the work of Glaiel, Moulton, and Madnick (2013) which utilizes system dynamics (a form of continuous SPSM) to describe and evaluate agile processes.

An example of an agile software development process is Scrum, which focuses around the use of a backlog (a priority queue of new features or user stories to be implemented in the software). Scrum utilizes the concept of a scrum to plan the next sprint (period of iterative development typically consisting of 45 days) (Schwaber and Beedle 2001). The sprint is then conducted with short meetings (known as stand-up meetings) held each day to identify the work that has been completed the previous day and the work to be completed during the current day. At the end of each sprint (or set of sprints) a release of the software occurs bringing the newest features to the user. Although the intention of agile is to facilitate faster development with a goal of higher quality, yet there is still a continual buildup of technical debt.

Technical debt is a metaphor originally coined by Ward Cunningham (Cunningham 1992) as a way of explaining the need to restructure software using a financial metaphor, for the benefit of management. Technical debt was later mentioned by Fowler et al. (1999) as an argument towards the benefits of refactoring as a means of reducing technical debt. Today, the agile community views the management of known and unknown technical debt items as first class objects that once identified, should be tracked (over their lifetime) as a part of a combined backlog (Gat and Heintz 2011). For a deeper exploration of recent research, we refer the reader to a comprehensive literature review by Tom, Aurum, and Vidgen (2013).

Technical Debt Management comprises the actions of identifying the debt and making decisions about which debt should be paid and when. The current industry focus has been on identifying and tracking debt as part of the working project backlog (Kruchten, Nord, and Ozkaya 2012). Essentially, we can see the creation of code smells within a code base akin to taking on debt (Fontana, Ferme, and Spinelli 2012), and the longer they are allowed to remain (without refactoring (Fowler, Beck, Brant, Opdyke, and Roberts 1999) (Neill and Laplante 2006)) the more influence they will have on the code base (Counsell, Hierons, Hamza, Black, and Durrand 2010) and project velocity (Power 2013). This influence resonates through the code and makes the software harder to extend and maintain in the future, thus causing software engineers to pay interest on the debt by increasing the amount of effort required to affect a change (Nugroho, Visser, and Kuipers 2011). These proposed strategies (which are the focus of this paper) represent a set of basic practices that can be applied by any company within the industry. There exist more advanced processes

such as basic cost benefit analysis (Seaman, Guo, Izurieta, Cai, Zazworka, Shull, and Vetro 2012), real options analysis, net-present value, and total cost of ownership (Sullivan, Chalasani, Jha, and Sazawal 1999), and portfolio approaches (Guo and Seaman 2011) which have also been suggested, but they are outside the scope of this work.

The purpose of this work is to utilize simulation as a method to evaluate the different strategies for basic technical debt management proposed in the literature (Kruchten, Nord, and Ozkaya 2012), (Letouzey and Ilkiewicz 2012), and(McConnell 2008), in the context of a Scrum based agile software development process. The proposed model is a discrete-event simulation and is based on the work of Glaiel, Moulton, and Madnick (2013) but which focuses on understanding the process of technical debt management rather than the agile process as a whole. The contribution of this work will be to confirm the concepts proposed in (Kruchten, Nord, and Ozkaya 2012), (Letouzey and Ilkiewicz 2012), and (McConnell 2008).

3 CONCEPTUAL MODEL

The model we have developed is designed to simulate the Scrum development process (Schwaber and Beedle 2001) from the perspective of the Product Owner (or manager in charge of a product). The conceptual model is depicted in Figure 1. In general, the development of the product is done in an iterative fashion. Each development iteration is called a sprint, wherein the work to be completed is finalized and 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 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 development of a system can be composed of several projects. In this study we limited the number of projects to 1.

Each project begins at the 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. Once complete the sprint velocity to determine where the process can be improved. Sprint velocity is basically a means to determine how on-track the development team was 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 difference in work completed over work assigned between two consecutive sprints. The same metric can be calculated for releases as well as 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 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 typically are no such processes which leads to technical debt in the released software.

This model consists of the *Work Items* (entities) and of the *Software Engineers* (resources).

Each work item has the following properties:

- **Engineer** the software engineer assigned to this work item.
- **Identifier** a unique identifier to track this work item.
- **Type** represents the type of work to be completed and is one of the set New Feature, Bug/Defect, or Technical Debt (Major Refactoring).

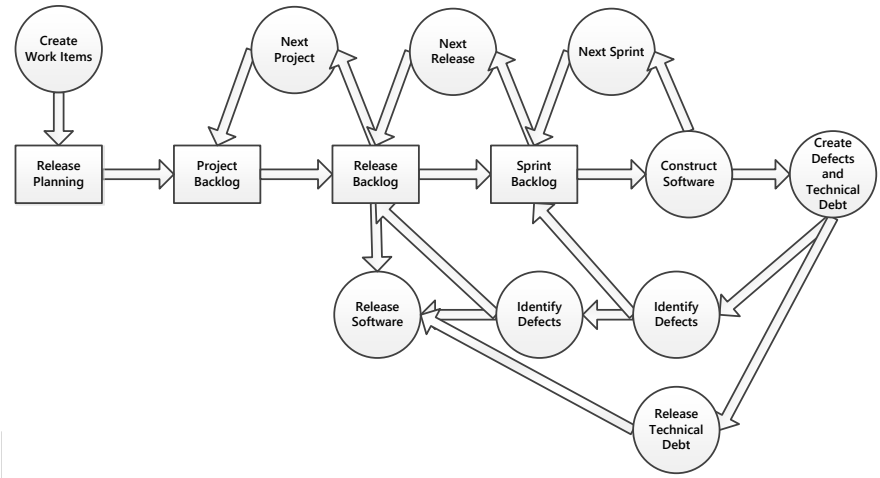

Figure 1: Conceptual model for a discrete-event simulation of the Scrum agile process which includes both defect and technical debt creation.

- **Size** which is an estimate of the change to the system and is measured in Source Lines of Code (SLOC). The size was set using a triangular distribution of TRIANG(250,100,2500) SLOC.
- **Effort** which is 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 (Moløkken-Østvold, Haugen, and Benestad 2008) (Tamrakar and Jørgensen 2012), the Delphi Approach (Rowe and Wright 1999), etc.) and has the base unit of hours. The effort was defined using a triangular distribution set at TRIANG(0.5, 1, 10) man-hours, for New Features and Technical Debt, while Defects were set to either TRIANG(3,8,24) or TRIANG(1,2,3) for major and minor defects, respectively.
- **Priority** which is 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 (Sharp, Finkelstein, and Galal 1999). We used a discrete distribution such that 25% are Priority 1 or Priority 2, 15% 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.

Each software engineer has the following properties:

- **Type** is 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** which is 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.

There are three backlogs (sets of work items to be process by the software engineers) which are defined as follows:

- **Project Backlog** is 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** is the master list of all work to be completed during a given release period, and it is ordered similar to the product backlog. The release backlog is further decomposed into one or more sprint backlogs.
- **Sprint Backlog** is the master list of all work to be completed during a given sprint, and is ordered similar to the product and release backlogs.
- Note: both the Release and Sprint backlogs have a maximum capacity (in hours) representing the total work that can be completed by a set of software engineers during a given time period.

3.1 The Simulation Process

The general simulation process can be seen in Figure 2 while the input parameters used for each of the models can be found in Table 1. The following is a narrative describing this process utilizing the above defined work items, software engineers, and backlogs.

Table 1: Input parameters, their descriptions and default values used during simulation.

Input	Description	Value
$MaxSprintEffort$	Maxmimum 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 for a project.	3 releases
$MaxProjects$	Maximum number of projects per simulation.	1 project
$InitialTD$	Initial amount of TD in the system.	1000 SLOC
$SprintDuration$	Maximum sprint length in days.	45 days
$SprintTDIteration$	How often a TD only sprint should occur.	Every 2nd Sprint
$SprintTDPercent$	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

The process begins with the creation of work items to be evaluated during release planning. Release Planning is essentially a limitless priority queue which stores these work items until a project is ready to commence. Once $MaxProjectEffort$ amount of work has accumulated, project planning commences. This then moves work items from Release Planning into the Project Backlog. Once the project backlog has reached a capacity to fill *MaxReleases* release backlogs a release begins.

A release begins by first incrementing the *currentRelease* variable. Here we determine if we can continue and if so 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, the sprint cycle is started. Within the sprint cycle the following occurs: First, the *CurrentSprint* variable is incremented and then the sprint backlog is filled to the 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

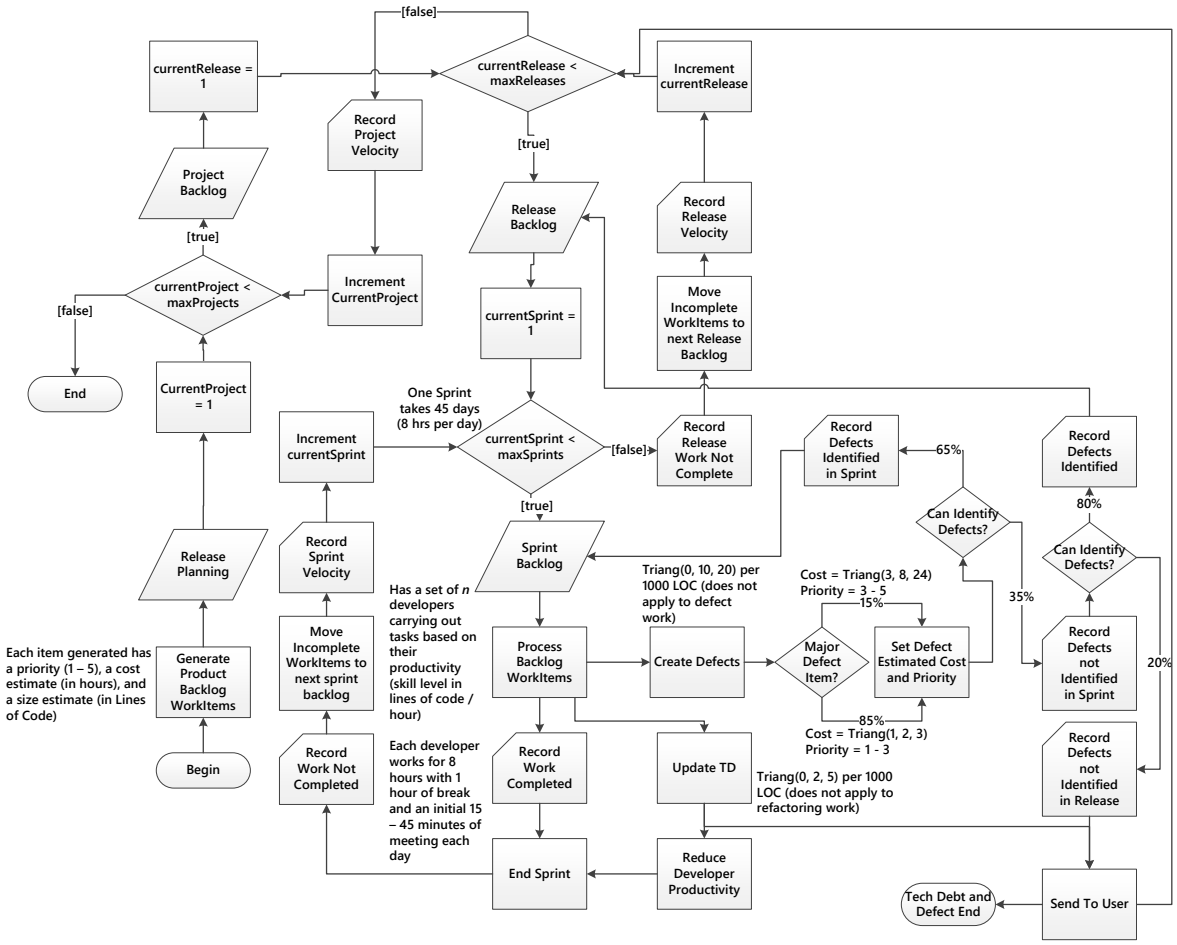

Figure 2: Diagram of the base model for the scrum software development process including defect and technical debt incorporation.

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 the sprint portion of the model, as the software engineers are completing the work items, there is the possibility that each completed work item can have defects or possibly can contain technical debt. In either case, the work items is still considered completed but at the same time the model generates new items for processing. In the case of defects (which are bugs in the software), based on industry average of 10 - 20 defects per 1000 SLOC (McConnell 2004), we will generate a new set of defects using the distribution TRIANG(0, 10, 20) (only for each technical debt or new feature work item processed). Approximately 85% of the defects occurring in a software product have been shown to be easily removed (taking 2 - 3 hours on average), while the remainder take much longer (from 3 hours to 3 days) (McConnell 2004). Given this, we assign a cost to the 85% (minor) new defects to be a value from TRIANG(1,2,3) with a priority between 1 and 3, and for the rest (major) we set the cost to a value from TRIANG(3, 8, 24) and a priority between 3 and 5.

Once the defects have been created we must determine whether or not testing and code reviews will catch the bugs. For simplicity, we assume in-sprint testing and reviews will catch approximately 65% of the defects and that 35% will escape and make it to the release test and review process. The caught defects will be put into the next sprint backlog. The defects that went to release testing will be caught with a 80% chance (due to more thorough testing and code review prior to release) and will be placed in the next release's backlog. Finally, the remaining defects will be output to the user.

As for the technical debt, we will generate TRIANG(0, 2, 5) 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 and go out to the users. It should be noted that for both the technical debt and defects generated we are tracking the identified and unidentified instances as variables of the system. We track technical debt specifically due to the impacts on software engineer productivity. 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:

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

Given this conceptual model we assume 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 units possible. For the base model we assume that technical debt is not a concern and that any refactorings are 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 (although we know that they are estimates only). Finally, we assume that the priority of the work items and their order in the list reflects the dependencies between them. That is, if a work item is dependent upon other work items, then those it depends on come before it in the backlog.

4 EXPERIMENTAL DESIGN

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

4.1 Experiments

Our experiments are designed to explore different methods of technical debt management which have been proposed in the literature. Given this we have identified four specific models which will be used for comparative analysis. The first model or base model is an implementation of the conceptual model and does not consider any means for technical debt management. The goal of the base model is to verify that the process is correct prior to evaluating the other approaches. The second model maintains a separate list of technical debt items which allows deliberate tracking of the technical debt items. The third and fourth models utilizes this list and continuously monitors the product for new instances of technical debt.

These middle two models can use either a percentage based or sprint based strategy to remove technical debt. In the percentage based method, a certain percent of work to be done in a sprint is the removal of technical debt while the rest is defect or new feature work. In the sprint-based method, an entire sprint of each release is devoted to the removal of technical debt. The final model is based on the concept of a technical debt threshold (McConnell 2008), which is built upon the active monitoring model and utilizes a threshold to identify when technical debt should be removed. This model has two possible threshold approaches: the first will begin technical debt removal once the current level has reach an upper threshold, and the other will utilize both an upper threshold a lower threshold to stop the technical debt removal phase.

Table 2: Summary of the experimental design and comparative analysis to be performed.

Model	Strategy	Acronym
Base	–	B
TD List	Percent	TDL-P
	Sprint	TDL-S
TD List with Active TDM	Percent	ATDM-P
	Sprint	ATDM-S
TD Thresholding	Upper Threshold Only	TDT-U
	Upper and Lower Threshold	TDT-UL

Using these models we intend to construct and compare the results of each model and the various strategies employed, in order to determine which technical debt management strategy is superior. A summary of these models can be found in Table 4. The comparative analysis design can be seen in Figure 3. Where we compare between strategies of each model and taking the best of this model we compare it to the best of the next model type. In each of these comparisons we are looking at the following five metrics: average sprint velocity (SV), average release velocity (RV), average accumulated technical debt (TD), number of items completed (WC), and effort expended for completed items (EC).

4.2 Data Generation

Utilizing existing theoretical concepts and models we plan to 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 (Cohn 2006) and (McConnell 2006). The size and cost/effort estimates for technical debt items will be based on the models identified in (Marinescu 2012b)(Marinescu 2012a), (Curtis, Sappidi, and Szynkarski 2012), and (Nugroho, Visser, and Kuipers 2011). The defects generated during the process follow the empirical models described in (McConnell 2004) which identifies the size and estimated effort required to remove these defects.

5 RESULTS AND ANALYSIS

In this study we conducted several simulations of the models described in the previous section. For each simulation we conducted a total of 8125 replications. This number of replications was used 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 metrics values for each metric of concern over the developed models can be found in Table 3.

In our study we conducted the following comparisons, which are further described in the following subsections: TDL-P vs. TDL-S, TDL-S vs. B, ATDM-P vs. ATDM-S, ATDM-S vs. TDL-S, TDT-UL vs. TDT-U, and ATDM-S vs. TDT-U. In each of the comparisons we assume the following null hypothesis: $H_0 : Model1_{metric} = Model2_{metric}$ and the following alternative hypothesis: $H_A : Model1_{metric} \neq Model2_{metric}$. The resulting mean differences and 95% confidence intervals for the comparison of mean metric values between contrasted models is provided in Table 4.

5.1 TDL-P vs. TDL-S

From the results in Tables 3 and 4 we are 95% confident that there is a significant difference between TDL-P and TDL-S RV and that TDL-P RV is significantly less than that of TDL-S. We are also 95% confident that there is no significant difference between the SV of these two models. Furthermore, we are 95% confident that the TDL-P model TD is significantly greater than that of the TDL-S. Finally, for

Table 3: Results for metrics of concern for each model after 8125 replications of the simulation.

Model	*RV*	*SV*	*TD*	*WC*	*EC*
B	37.7306 ± 0.05	12.8446 ± 0.01	19088.02 ± 53.94	13662.7 ± 26.53	4098.61 ± 8.01
TDL-S	37.7835 ± 0.04	12.854 ± 0.01	15743.76 ± 24.91	13682.12 ± 24.91	4104.01 ± 7.54
TDL-P	37.605 ± 0.07	12.8323 ± 0.02	15867.96 ± 45.63	13639.01 ± 29.23	4090.83 ± 8.81
ATDM-P	34.2469 ± 0.22	12.4163 ± 0.05	14965.51 ± 93.83	12895.24 ± 69.78	3869.27 ± 20.98
ATDM-S	25.5887 ± 0.37	9.6881 ± 0.12	7742.4 ± 69.39	10260.57 ± 97.78	3079.66 ± 29.28
TDT-U	37.7874 ± 0.04	12.8493 ± 0.01	15950.73 ± 40.94	13683.27 ± 25.07	4105.57 ± 7.6
TDT-UL	37.7878 ± 0.04	12.8495 ± 0.01	19129.51 ± 51.96	13683.77 ± 25.07	4104.79 ± 7.57

both WC and EC we are 95% confident that the difference is significant and that the values for TDL-P are greater than those of TDL-S.

Thus, the use of the percentage based approach pales in comparison to the TD-only sprint based approach. Although, these results indicate a statistically significant but practically minor gain in using the TD-only sprint based approach. It is possible that if a given team was to fine-tune the *SprintTDPercentage* of using historical analysis then the percentage based approach would likely equal or exceed the TD-only sprint based approach.

5.2 TDL-S vs. B

Given that the TDL-S model prevailed in comparison to the TDL-P model, we then compared this to the base model to identify the overall gain in performance that maintaining a list of technical debt items provides. We are 95% confident that there is no significant difference between metric values for the TDL-S and B models, with the exception of TD. In the latter case, we are 95% confident that TD of the TDL-S model is less than that of the B model.

These results are as expected. That is, since the TDL-S (and TDL-P) model simply adds a technical debt list on top of the base model and begins using this as a work items supply into the sprint backlog, it is straight-forward that the amount of accumulated technical debt would decrease as items from the technical debt list are processed. This result confirms the proposed technical debt combined backlog (Kruchten, Nord, and Ozkaya 2012) and technical debt list (McConnell 2008) strategies from the literature.

5.3 ATDM-P vs. ATDM-S

The next comparison is between the ATDM-P and ATDM-S models. In this comparison we found that, for all metrics we are 95% confident that the value of ATDM-P metric values are significantly greater than the corresponding values from the ATDM-S model.

This result can be interpreted as follows: The use of a percentage based approach will yield slightly higher sprint and release velocities while also increasing the amount of work done, which is excellent. Yet, when we look at the width of the confidence interval for accumulated technical debt we see that this increase in work completed comes at the price of significant technical debt overhead. From this we must conclude that the minor increases in production are not worth the loss in technical debt and hence the ATDM-S model is the better choice.

5.4 ATDM-S vs. TDL-S

Taking the ATDM-S model from the previous comparison we now compare it to the best model of the previous step in the model hierarchy, TDL-S. As can be seen in Table 4, a similar effect occurs as in the ATDM-P vs. ATDM-S comparison. Hence, we are 95% confident that for all metrics of concern the difference between the metric values are significant, and for each metric the TDL-S model's values are significantly greater than those of the ATDM-S.

Table 4: Confidence interval results from model comparisons at the 95% confidence level.

Comparison	*RV*		*SV*	
	Mean	Confidence Interval	Mean	Confidence Interval
TDL-P vs TDL-S	-0.1785	(-0.2561, -0.1009)	-0.0217	(-0.04674, 0.0033)
TDL-S vs B	0.0529	(-0.00195, 0.1077)	0.0094	(-0.008309, 0.02711)
ATDM-P vs ATDM-S	8.6582	(8.2492, 9.0672)	2.7282	(2.6023, 2.8541)
ATDM-S vs TDL-S	-12.1948	(-12.579, -11.8106)	-3.1659	(-3.2905, -3.0413)
TDT-U vs TDT-UL	0.0004	(-0.03421, 0.035)	0.0002	(-0.00027, 0.00067)
ATDM-S vs TDT-U	-12.1987	(-12.5797, -11.8177)	-3.1612	(-3.2846, -3.03784)

Comparison	*WC*		*EC*	
	Mean	Confidence Interval	Mean	Confidence Interval
TDL-P vs TDL-S	-13.18	(-25.4533, -0.9067)	-43.11	(-8.3903, -2.317)
TDL-S vs B	5.4	(-5.2187, 16.0187)	19.42	(-15.726, 54.5657)
ATDM-P vs ATDM-S	789.61	(756.4227, 822.7973)	2634.67	(2524.011, 2745.33)
ATDM-S vs TDL-S	-1024.35	(-1055.31, -993.39)	-3421.55	(-3524.81, -3318.29)
TDT-U vs TDT-UL	-0.78	(-9.835, 8.275)	0.5	(-42.05, 43.05)
ATDM-S vs TDT-U	-1025.91	(-1056.27, -995.55)	-3422.7	(-3528.39, -3317.001)

Comparison	*TD*	
	Mean	Confidence Interval
TDL-P vs TDL-S	124.2	(62.7518, 185.6482)
TDL-S vs B	-3344.26	(-3406.35, -3282.17)
ATDM-P vs ATDM-S	7223.11	(7120.67, 7325.55)
ATDM-S vs TDL-S	-8001.36	(-8083.824, -7918.896)
TDT-U vs TDT-UL	3178.78	(3119.26, 3238.298)
ATDM-S vs TDT-U	-8208.33	(-8287.47, -8129.187)

These results can be interpreted similarly as above. That is, the TDL-S model does complete more work with more effort, and it also shows that the velocity (ability to increase production between sprints and releases) does increase. Unfortunately, the downside is that there is nearly twice the amount of technical debt in the TDL-S over the ATDM-S model (see Table 3). Since the ATDM-S model is built on top of the TDL-S model, the differences in the ATDM-S model can only be attributed to the increased technical debt detected using the automated monitoring capabilities. This result confirms the active monitoring and removal strategy from the literature.

5.5 TDT-U vs. TDT-UL

The final inter-strategy comparison is between the two forms of threshold applied on top of the ATDM-P model. As can be seen in Table 4 we are 95% confident that there is no significant difference in the contrasting models' metric values, excluding TD. As for TD we are 95% confident that a significant difference exists such that the TDT-UL model accumulates more debt than the TDT-U model.

This result is unexpected since as seen by the mean values and half-widths the overall mean accumulated technical debt is always above the upper threshold of 5000 man-hours for TDT-U and TDT-UL and above the lower threshold of 1000 man-hours for the TDT-UL model. Given this, further exploration must be conducted to determine why this occurred. The interpretation of these results would be to conclude that unless the accumulated technical debt value reaches the lower threshold the models are essentially

equivalent, but given the discrepancy in accumulated technical debt we consider the TDT-U model to be currently superior. This brings us to the final comparison between the ATDM-S and TDT-U models.

5.6 ATDM-S vs. TDT-U

The final comparison is between the superior models of the last two categories of strategies: ATDM-S and TDT-U. From Table 4 we can see that for all metric values with 95% confidence there is a significant difference between these models and this difference leads to ATDM-S having lower values than TDT-U in every case. This is not surprising since TDT-U is a direct descendant of the ATDM-P model and so in comparison to ATDM-S the interpretation is the same.

6 CONCLUSION

In this paper 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 conducting technical debt remediation sprints is the superior combination of techniques. 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 (such as in the TD-only sprint), this deserves further investigation and will be looked into as a part of immediate future work.

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. As for future work we intend to continue to explore various combinations as well as to 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.

REFERENCES

Cohn, M. 2006. *Agile estimating and planning*. Prentice Hall.

Counsell, S., R. M. Hierons, H. Hamza, S. Black, and M. Durrand. 2010. “Is a strategy for code smell assessment long overdue?”. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, WETSoM ’10, 3238. Cape Town, South Africa: ACM.

Cunningham, W. 1992, December. “The WyCash portfolio management system”. *SIGPLAN OOPS Mess.* 4 (2): 2930.

Curtis, B., J. Sappidi, and A. Szynkarski. 2012, June. “Estimating the size, cost, and types of Technical Debt”. In *Managing Technical Debt (MTD), 2012 Third International Workshop on*, 49–53.

Fontana, F., V. Ferme, and S. Spinelli. 2012, June. “Investigating the impact of code smells debt on quality code evaluation”. In *Managing Technical Debt (MTD), 2012 Third International Workshop on*, 15–22.

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

Gat, I., and J. D. Heintz. 2011. “From assessment to reduction: how cutter consortium helps rein in millions of dollars in technical debt”. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD ’11, 2426. Waikiki, Honolulu, HI, USA: ACM.

Glaiel, F., A. Moulton, and S. Madnick. 2013. “Agile Project Dynamics: A System Dynamics Investigation of Agile Software Development Methods”.

Guo, Y., and C. Seaman. 2011. "A portfolio approach to technical debt management". In *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD '11, 3134. Waikiki, Honolulu, HI, USA: ACM.

Kellner, M. I., R. J. Madachy, and D. M. Raffo. 1999. "Software process simulation modeling: why? what? how?". *Journal of Systems and Software* 46 (2): 91–105.

Kruchten, P., R. L. Nord, and I. Ozkaya. 2012, December. "Technical Debt: From Metaphor to Theory and Practice". *Software, IEEE* 29 (6): 18–21.

Letouzey, J., and M. Ilkiewicz. 2012, December. "Managing Technical Debt with the SQALE Method". *Software, IEEE* 29 (6): 44–51.

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

Marinescu, R. 2012a. "Assessing and Improving Object-Oriented Design".

Marinescu, R. 2012b, October. "Assessing technical debt by identifying design flaws in software systems". *IBM Journal of Research and Development* 56 (5): 1–13.

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

McConnell, S. 2006. *Software Estimation: Demystifying the Black Art*. Microsoft Press.

McConnell, S. 2008. "Managing Technical Debt". Best Practices White Paper 1, Construx.

Moløkken-Østvold, K., N. C. Haugen, and H. C. Benestad. 2008. "Using planning poker for combining expert estimates in software projects". *Journal of Systems and Software* 81 (12): 2106–2117.

Neill, C., and P. Laplante. 2006. "Paying down design debt with strategic refactoring". *Computer* 39 (12): 131–134.

Nugroho, A., J. Visser, and T. Kuipers. 2011. "An empirical model of technical debt and interest". In *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD '11, 18. Waikiki, Honolulu, HI, USA: ACM.

Power, K. 2013. "Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options". In *Managing Technical Debt (MTD), 2013 4th International Workshop on*, 28–31.

Ramasubbu, N., and C. F. Kemerer. 2013. "Towards a model for optimizing technical debt in software products". In *Managing Technical Debt (MTD), 2013 4th International Workshop on*, 51–54.

Rowe, G., and G. Wright. 1999. "The Delphi technique as a forecasting tool: issues and analysis". *International Journal of Forecasting* 15 (4): 353–375.

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

Seaman, C., Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, and A. Vetro. 2012, June. "Using technical debt data in decision making: Potential decision approaches". In *Managing Technical Debt (MTD), 2012 Third International Workshop on*, 45–48.

Sharp, H., A. Finkelstein, and G. Galal. 1999. "Stakeholder identification in the requirements engineering process". In *Database and Expert Systems Applications, 1999. Proceedings. Tenth International Workshop on*, 387–391.

Sullivan, K. J., P. Chalasani, S. Jha, and V. Sazawal. 1999. *Software design as an investment activity: a real options perspective*. Risk Books.

Tamrakar, R., and M. Jørgensen. 2012. "Does the use of Fibonacci numbers in Planning Poker affect effort estimates?".

Tom, E., A. Aurum, and R. Vidgen. 2013, June. "An exploration of technical debt". *Journal of Systems and Software* (86:6): 1498–1516.

Zhang, H., B. Kitchenham, and D. Pfahl. 2008, December. "Software Process Simulation Modeling: Facts, Trends and Directions". In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, 59–66.