Proceedings of the 2014 Winter Simulation Conference
A. Tolk, S. D. Diallo, I. O. Ryzhov, L. Yilmaz, S. Buckley, and J. A. Miller, eds.

A SIMULATION STUDY OF PRACTICAL METHODS FOR TECHNICAL DEBT MANAGEMENT IN AGILE SOFTWARE DEVELOPMENT

Isaac Griffith
Clemente Izurieta

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

Hanane Taffahi
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 and validation of proposed methods for technical debt management in the context of agile development. In practice, such evaluations are either too costly or too time consuming to be conducted using traditional empirical methods. In this paper, we describe a set of simulations based on models of the agile development process, Scrum, and the integration of technical debt management. The purpose of this study is to identify which strategy is superior and to provide empirical evidence to support existing claims. The models presented are based upon conceptual and industry models concerning defects and technical debt. The results of the simulations provide compelling evidence for current technical debt management strategies proposed in the literature that can be immediately applied by practitioners.

1 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 (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), 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 (Schwaber and Beedle 2001).

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 used in this study. Section 5 describes the results and analysis. 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 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 (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 literature, 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 identified the following other less used simulation modeling paradigms: qualitative simulation, knowledge(rule)-based simulation, role-playing games, agent-based simulation, discrete-time simulation, stigmergy theory, and emergent/unbound systems. They looked into the levels at which simulation is used in software process modeling noting three levels of abstraction (based on relationships modeled): *system level, process level,* and *entity level.* At the system level, the process is modeled as a system defined by external parameters which are allowed to vary continuously over time. The process level delves deeper than the causal relationships of the system level and describes the process as a set of entities, resources, and the relationships between them. The entity level is more in depth look at the process focusing on the actual entities (e.g., software engineers) and the tasks they need to complete in the context of the process. They 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 (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 et al. (2013) which utilizes system dynamics (a form of continuous SPSM) to describe and evaluate agile processes.

Scrum is an agile development process 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 sprint planning meeting (constrained to an 8 hour time limit) 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 or scrums and constrained to 15 minutes in duration) 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, there is still a continual buildup of technical debt.

Technical debt is a metaphor originally coined by Ward Cunningham (1992) as a way of explaining the need to restructure software using a financial metaphor, for the benefit of management. Fowler et al. (1999) suggests (as an argument towards the benefits of refactoring) that reducing technical debt should motivate development teams to practice constant refactoring. Current research in the agile community views the management of known and unknown technical debt as first class objects that once identified, should be tracked (over their lifetime) as a part of a combined backlog (Gat and Heintz 2011) or a separate technical debt list (Seaman and Guo 2011). Recently, Schmid has developed a formal framework which divides technical debt into potential technical debt (any item which may be identified as TD) and

effective technical debt (those potential technical debt items which are of concern and slated for remediation) (Schmid 2013). 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 identification, assessment, and remediation of technical debt throughout a software system. The current industry focus has been on identifying and tracking debt as part of the working project backlog (Kruchten, Nord and Ozkaya 2012). An example of such technical debt items are code smells, which are poorly designed areas of the software which strongly indicate a need for refactoring (Fowler, et al. 1999). Essentially, we can interpret the creation of code smells within a software system as taking on debt (Fontana, Ferme and Spinelli 2012), and the longer they are allowed to remain (without refactoring) (Fowler, et al. 1999; Neill and Laplante 2006) the more influence they will have on the code base (Counsell, et al. 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). The 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 are more advanced processes such as basic cost benefit analysis (C. Seaman, et al. 2012), real options analysis, net-present value, and total cost of ownership (Sullivan, et al. 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 present 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; McConnell, Managing Technical Debt 2008). We utilize an agile development context, Scrum, and show that technical debt management strategies can be evaluated. The proposed models are evaluated using discrete-event simulation based on the work of Glaiel et al. (2013) but focusing on understanding the process of technical debt management rather than the agile process as a whole. The contributions of this work confirm the concepts proposed in the literature and the use of simulation as a means to help managers evaluate their own TDM strategies.

3 CONCEPTUAL MODEL

The model we have developed is designed to simulate the Scrum development process (Schwaber and Beedle 2001), as depicted in Figure 1, 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.

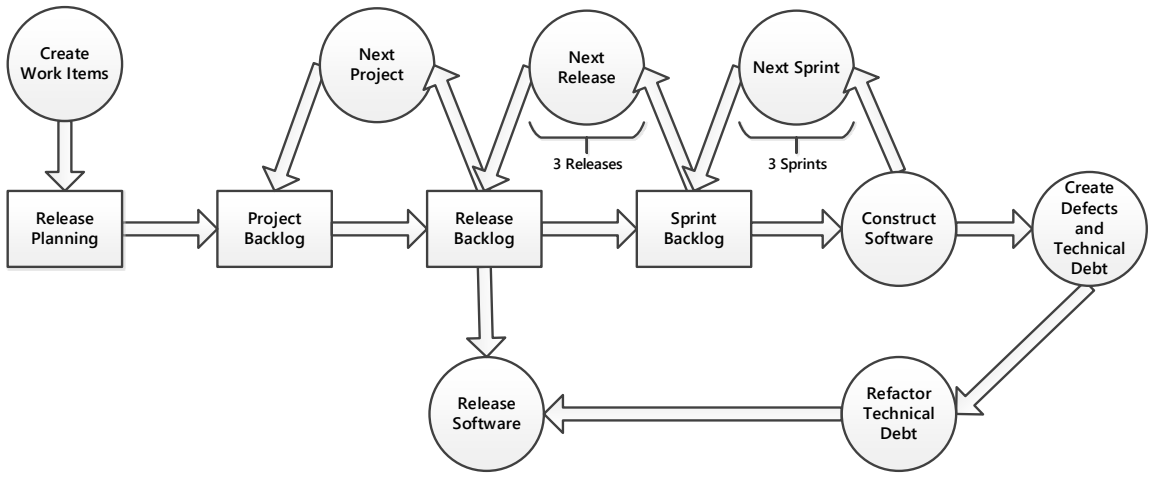

Figure 1. Conceptual model for a discrete-event simulation of the Scrum agile process which includes both defect and technical debt creation.

Table 1. 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 (Sharp, Finkelstein and Galal 1999). Represented as 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.
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 (Moløkken-Østvold, Haugen and Benestad 2008; Tamrakar and Jørgensen 2012), the Delphi Approach (Rowe and Wright 1999), etc.). The effort is set using a triangular distribution TRIANG(0.5, 1, 10)[1], 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(250,100,2500).
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 2. 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.

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 rele ases 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.

Table 3. 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.

The conceptual model consists of three types of objects: Work Items, Software Engineers, and Backlogs. Each work item has the attributes described in Table 1. Each software engineer has the attributes defined in Table 2. Each of the backlogs consists of the properties defined in Table 3.

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. 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 work completed over work assigned between two consecutive sprints. The same metric can be calculated for re leases 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

Table 4. 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

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.

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 ca n be found in Table 4. 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

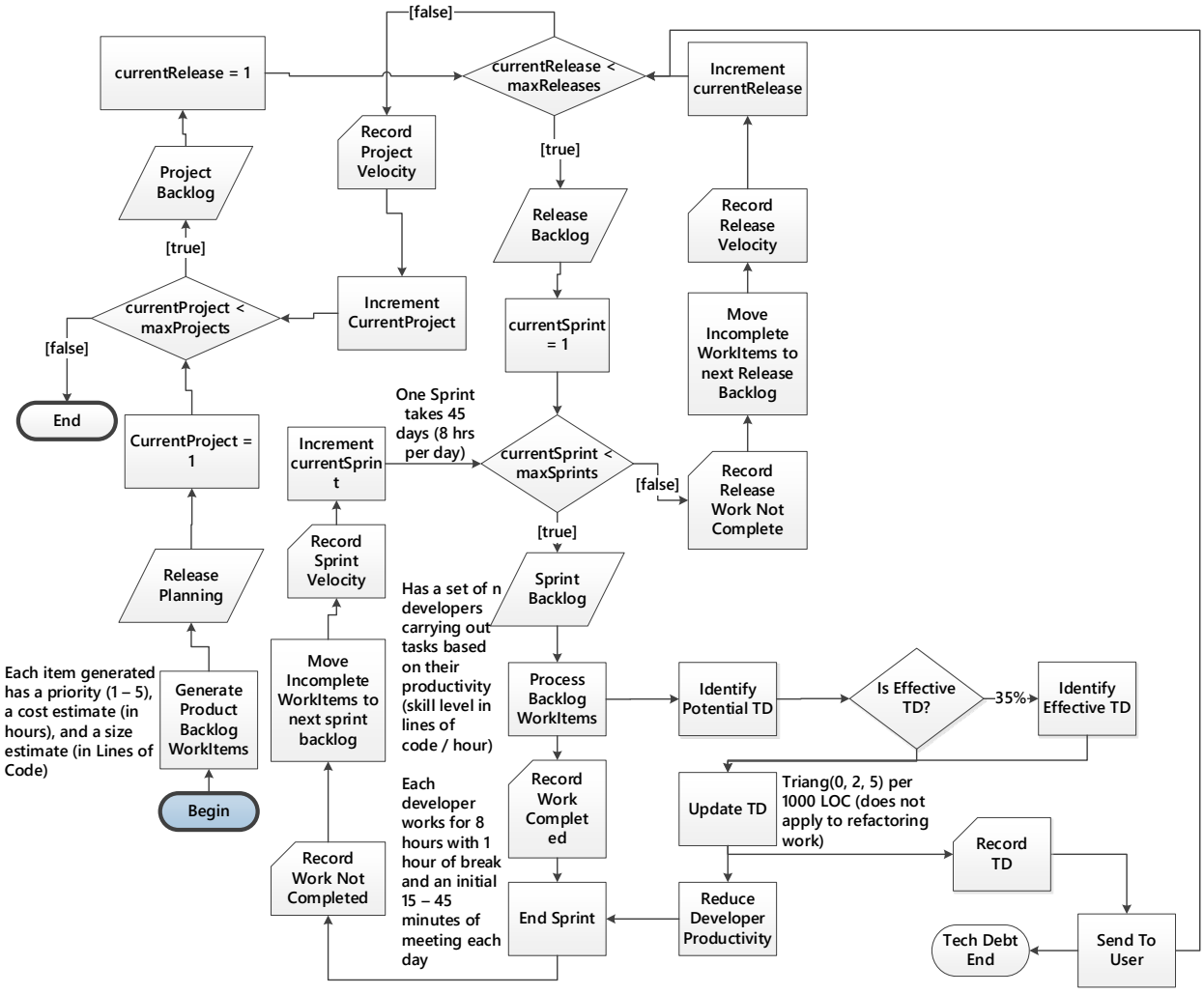

Figure 2. Diagram of the base model for the scrum software development process including defect and technical debt incorporation.

is started. Within the 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 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.

4 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.

Table 5. 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**

4.1 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 (McConnell, Managing Technical Debt 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 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 5.

4.2 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 (Cohn 2006) and (McConnell, Software Estimation: Demystifying the Black Art 2006). The size and cost/effort estimates for technical debt items are based on the models identified in (Marinescu 2012), (Curtis, Sappidi and Szynkarski 2012), and (Nugroho, Visser and Kuipers 2011). The defects generated during the process follow the empirical models described in (McConnell, Code Complete: A Practical

Table 6. Summary of the models and strategies developed for 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	65.51860656
TDL-P vs Base	-2536.8	1393.292	-528.332	-2310.236	-1921.886531
ATDM-S vs ATDM-P	-645.264	350.604	-137.416	-617.648	-556.3476327
ATDM-S vs TDL-P	-548.724	420.008	-105.564	-506.408	-462.0379918
TDT-U vs TDT-UL	2662.508	-1369.668	548.176	2325.976	1959.103512
ATDM-S vs TDT-U	-2565.968	1439.072	-518.784	-2220.152	-1869.516664
TDL-P vs ATDM-P	125.708	23.624	19.844	15.74	37.2169801

Handbook of Software Construction 2004) which identifies the size and estimated effort required to remove these defects.

5 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 metrics values for each metric of concern over the developed models can be found in Table 6. Figure 3 depicts the mean metric values (excluding WC) between simulations, while Figure 4 depicts the change in CC, WC, and TD across simulations.

Each comparison, whose values are shown in Table 6, 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 that the percentage based approach was superior. The reasoning behind this is that the percentage based results showed that more work items were 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

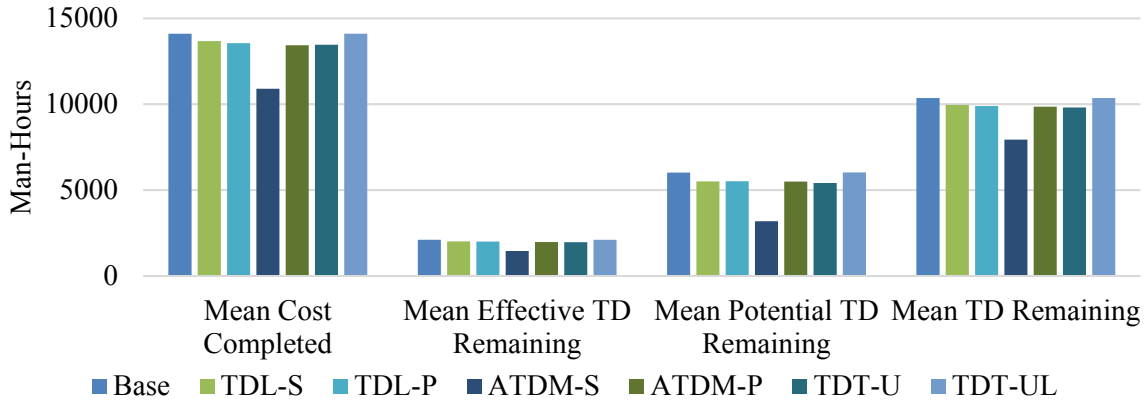

Figure 3. Comparison of metrics across simulations.

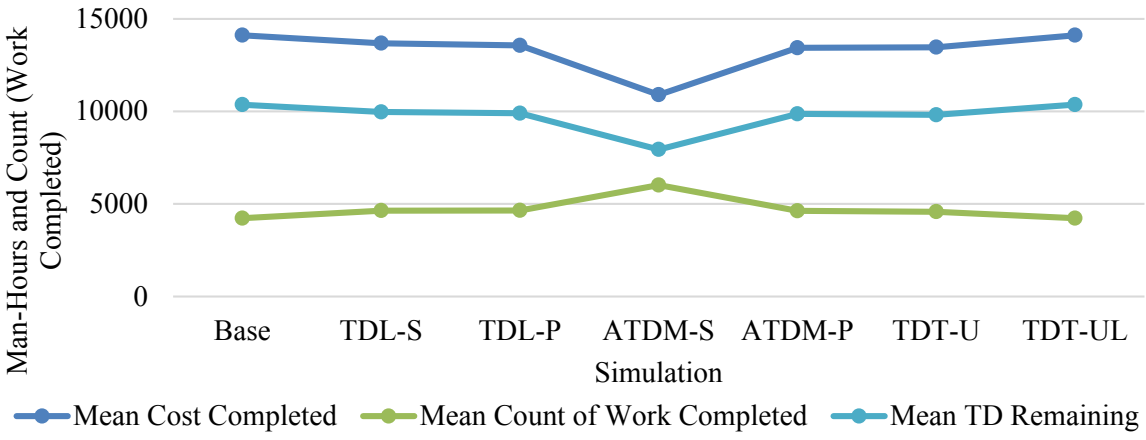

Figure 4. Change in work completed, technical debt remaining and mean cost completed across simulations

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.

6 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 nth 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.

REFERENCES

Cohn, Mike. 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?" *Proceedings of the 2010 {ICSE} Workshop on Emerging Trends in*

Software Metrics. Cape Town, South Africa: ACM. 32-38. http://doi.acm.org/10.1145/1809223.1809228.

Cunningham, Ward. 1992. "The WyCash Portfolio Management System." *SIGPLAN OOPS Mess.* 4 (2): 29-30. http://doi.acm.org/10.1145/157710.157715.

Curtis, B., J. Sappidi, and A. Szynkarski. 2012. "Estimating the Size, Cost, and Types of Technical Debt." *Managing Technical Debt (MTD), 2012 Third International Workshop on.* 49-53.

Fontana, F.A., V. Ferme, and S. Spinelli. 2012. "Investigating the Impact of Code Smells Debt on Quality Code Evaluation." *Managing Technical Debt (MTD), 2012 Third International Workshop on.* 15-22.

Fowler, Martin, Kent Beck, J Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Programs.* Addison-Weseley.

Gat, Israel, and John D. Heintz. 2011. "From Assessment to Reduction: How Cutter Consortium Helps Rein in Millions of Dollars in Technical Debt." *Proceedings of the 2nd Workshop on Managing Technical Debt.* Waikiki, Honolulu, HI, USA: ACM. 24-26. http://doi.acm.org/10.1145/1985362.1985368.

Glaiel, Firas, Allen Moulton, and Stuart Madnick. 2013. "Agile Project Dynamics: A System Dynamics Investigation of Agile Software Development Methods."

Guo, Yuepu, and Carolyn Seaman. 2011. "A Portfolio Approach to Technical Debt Management." *Proceedings of the 2nd Workshop on Managing Technical Debt.* Waikiki, Honolulu, HI, USA: ACM. 31-34. http://doi.acm.org/10.1145/1985362.1985370.

Kellner, Marc I, Raymond J Madachy, and David M Raffo. 1999. "Software Process Simulation Modeling: Why? What? How?" *Journal of Systems and Software* 46 (2): 91-105.

Kruchten, Philippe, Robert L. Nord, and Ipek Ozkaya. 2012. "Technical Debt: From Metaphor to Theory and Practice." *Software, IEEE* 29 (6): 18-21.

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

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

Marinescu, R. 2012. "Assessing Technical Debt by Identifying Design Flaws in Software Systems." *IBM Journal of Research and Development* 56 (5): 9:1-9:13.

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

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

—. 2006. *Software Estimation: Demystifying the Black Art.* Microsoft Press.

Moløkken-Østvold, Kjetil, Nils Christian Haugen, and Hans Christian Benestad. 2008. "Using Planning Poker for Combining Expert Estimates in Software Projects." *Journal of Systems and Software* 81 (12): 2106-2117. http://www.sciencedirect.com/science/article/pii/S0164121208000885.

Neill, C.J., and P.A. Laplante. 2006. "Paying Down Design Debt with Strategic Refactoring." *Computer* 39 (12): 131-134.

Nugroho, Ariadi, Joost Visser, and Tobias Kuipers. 2011. "An Empirical Model of Technical Debt and Interest." *Proceedings of the 2nd Workshop on Managing Technical Debt.* Waikiki, Honolulu, HI, USA: ACM. 1-8. http://doi.acm.org/10.1145/1985362.1985364.

Power, Ken. 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." *Managing Technical Debt (MTD), 2013 4th International Workshop on.* 28-31.

Ramasubbu, Narayan, and Chris F. Kemerer. 2013. "Towards a Model for Optimizing Technical Debt in Software Products." *Managing Technical Debt (MTD), 2013 4th International Workshop on.* 51-54.

Rowe, Gene, and George Wright. 1999. "The Delphi Technique as a Forecasting Tool: Issues and Analysis." *International Journal of Forecasting* 15 (4): 353-375. http://www.sciencedirect.com/science/article/pii/S0169207099000187.

Schmid, Klaus. 2013. "A Formal Approach to Technical Debt Decision Making." *Proceedings of the 9th international {ACM} Sigsoft conference on Quality of software architectures.* New York, NY, USA: ACM. 153-162. http://doi.acm.org/10.1145/2465478.2465492.

Schwaber, Ken, and Mike Beedle. 2001. *Agile Software Development with Scrum.* 1. Upper Saddle River, New Jersey: Prentice Hall.

Seaman, C., Yuepu Guo, C. Izurieta, Yuanfang Cai, N. Zazworka, F. Shull, and A. Vetro. 2012. "Using Technical Debt Data in Decision Making: Potential Decision Approaches." *Managing Technical Debt (MTD), 2012 Third International Workshop on.* 45-48.

Seaman, Carolyn, and Yuepu Guo. 2011. "Measuring and Monitoring Technical Debt." *Advances in Computers* 82: 25-46.

Sharp, H., A. Finkelstein, and G. Galal. 1999. "Stakeholder Identification in the Requirements Engineering Process." *Database and Expert Systems Applications, 1999. Proceedings. Tenth International Workshop on.* 387-391.

Sullivan, Kevin J, Prasad Chalasani, Somesh Jha, and Vibha Sazawal. 1999. *Software Design as an Investment Activity: A Real Options Perspective.* Risk Books.

Tamrakar, Ritesh, and Magne Jørgensen. 2012. "Does the Use of Fibonacci Numbers in Planning Poker Affect Effort Estimates?"

Tom, Edith, Aybüke Aurum, and Richard Vidgen. 2013. "An Exploration of Technical Debt." *Journal of Systems and Software* (0). http://www.sciencedirect.com/science/article/pii/S0164121213000022.

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

AUTHOR BIOGRAPHIES

ISAAC GRIFFITH is a PhD student in the Department of Computer Science at Montana State University in Bozeman, MT with Clemente Izurieta as his advisor. He holds a B.S. in Computer Science and a B.A. in Philosophy from Montana State University. His email address is Isaac.griffith@msu.montana.edu.

HANANE TAFFAHI is a MS student in the Department of Mechanical and Industrial Engineering at Montana State University in Bozeman, MT. She holds a B.S. in Engineering Management from Al Akhawayn University in Ifrane. Her email address is hanane.taffahi@msu.montana.edu.

DAVID CLAUDIO is an assistant professor of industrial engineering in the Department of Mechanical and Industrial Engineering at Montana State University, Bozeman, Montana. He received his Ph.D. in Industrial Engineering from the Pennsylvania State University. His research interests include Human Factors, Service Systems, Healthcare Engineering, and Decision Making. His email address is: david.claudio@ie.montana.edu.

CLEMETE IZURIETA is an Assistant Professor in the Computer Science department at Montana State University. Born in Santiago, Chile. His research interests include empirical software engineering, design and architecture of large software systems, design patterns, technical debt, the measurement of software quality and ecological modelling. Dr. Izurieta has approximately 16 years of experience working for various R&D labs at Hewlett Packard and Intel Corporation. His email address is clemente.izurieta@cs.montana.edu.