Managing Technical Debt in Timed-boxed Software Processes: Quantitative Evaluations

Luigi Lavazza, Sandro Morasca and Davide Tosi

Dipartimento di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria
21100 Varese, Italy
Email: luigi.lavazza@uninsubria.it
sandro.morasca@uninsubria.it
davide.tosi@uninsubria.it

***Abstract*—Technical debt is currently receiving great attention from researchers, because it is believed to affect software development to a great extent. However, it is not yet clear how technical debt should be managed. This is specifically true in time-boxed development processes (e.g., in agile processes organized into development sprints of fixed duration), where it is possible to remove technical debt as soon as it is discovered, or wait until the debt reaches a given threshold, or wait until a whole sprint can be dedicated to technical debt removal, etc. We aim at investigating the effectiveness of different technical debt management strategies and the consequences of a wrong perception of the actual technical debt. We are interested in the consequences on both the amount of functionality and the quality of the delivered software. We propose a System Dynamics model that supports the simulation of various scenarios in time-boxed software development and maintenance processes. The proposed model is conceived to highlight the consequences of management decisions. The proposed model shows how productivity and product quality depend on the way technical debt is managed. Our study shows that different strategies for managing technical debt in a time-boxed development and maintenance process may yield different results—in terms of both productivity and delivered software quality—depending on a few conditions. Software project managers can use customized System Dynamics models to optimize the development and maintenance processes, by making the proper decisions on when to carry out maintenance dedicated to decreasing the technical debt, and how much effort should be devoted to such activities.**

***Keywords*–Technical debt; System Dynamics; Simulation; Technical debt management; Software project management.**

I. Introduction

Both practitioners and researchers are dedicating a growing amount of attention to technical debt (TD). In general, TD is connected with a lack of quality in the code. The idea is that, if maintaining a piece of software of "ideal" quality has a given cost, maintaining a piece of software of "less than ideal" quality implies an extra cost.

It is also common knowledge that if no action is performed to improve code quality, a sequence of maintenance interventions will decrease quality, that is, TD increases and the cost of maintenance increases as well. Not managing TD at all could lead to code that is not maintainable.

However, it is easy to realize that too much time and effort dedicated to TD removal activities could have a negative effect on the overall speed of development, since time and effort devoted to TD management are usually subtracted to 'regular' development activities (developing new code, applying requirements changes, testing, etc.). So, a project manager should look for the optimal tradeoff between TD removal and regular development.

These considerations show that project managers need to identify the best TD management strategies and methods, and evaluate their effectiveness before putting them in practice. Quite importantly, managers need to reason in quantitative terms, in order to maximize the amount of released functionality at the best reasonable quality level.

For this purpose, we proposed a System Dynamics model that represents the development of software via a sequence of time-boxed development phases (e.g., Scrum sprints) [1]. Like any System Dynamics model, the proposed model can be simulated, thus providing quantitative indications concerning the effectiveness of development in terms of amount and quality of code delivered. The proposed model [1] was used to illustrate a few development scenarios and the consequences of TD and the adopted TD management practices. It was shown that dedicating a fixed fraction f of the available effort to TD remediation is more or less effective—with respect to both the delivered amount of functionality and the resulting software quality—depending on the value of f. In other words, to obtain good results, it is critical that the project manager guesses the value of f that optimizes the quantity and quality of delivered code at the beginning of the process. This is quite difficult, in general. On the contrary, it is easier to dedicate to TD removal a quantity of effort that is proportional to the size of the debt; it is also quite effective, with respect to both the delivered amount of functionality and the resulting software quality.

However, we can note that to implement the latter strategy, one has to know the amount of TD that is associated with the code. Generally, the amount of TD is evaluated by software managers via tools that perform static analysis of code and apply "expert" rules that detect the presence or absence of specific situations that contribute to the TD. Quite often, software managers do not have the time or the technical knowledge needed to verify the measure of TD provided by tools, hence they decide how much effort will be dedicated to TD removal based on the indications from tools.

Now, the indications from tools could overestimate (respectively, underestimate) the amount of TD. As a consequence, the project manager could dedicate more (respectively less) effort than needed to manage TD: we expect that in such cases the results—in terms of amount of delivered functionality and quality—will worsen, with respect to the situation when the proper amount of effort is dedicated to TD management.

Accordingly, in this paper we enhance the results presented in [1] by exploring the effects of basing decisions concerning TD management on an inaccurate perception of the amount of TD in the code.

The paper is organized as follows. In Section II, we provide background concerning TD and System Dynamics. In Section III, we introduce our model of software development and maintenance, characterized by time-boxed incremental phases. In Section IV, the model is used to simulate the behavior of the process when different strategies for allocating effort to pay off the TD are used. In Section V, we discuss the outcomes of simulations, especially as far as productivity and delivered quality are concerned. In Section VI the model is enhanced to take into account errors in the perception of TD, and the results of simulations are reported. Section VII accounts for related work. Finally, in Section VIII, we draw some conclusions and outline future work.

II. Background

We here concisely recall the TD concepts proposed in the literature that we later model and illustrate via simulations (Section II-A) and the principles of System Dynamics modeling (Section II-C).

A. Technical Debt

In the last few years, TD has received great attention from researchers. For example, a recent Systematic Mapping Study on TD and TD management (TDM) covering publications from 1992 and 2013 detected 94 primary studies to obtain a comprehensive understanding on the TD concepts and an overview on the current state of research on TDM [2].

An updated Systematic Mapping Study identified elements that are considered by researchers to have an impact on TD in the industrial environment [3]. The authors classified these twelve elements in three main categories: (1) Basic decision making factors, (2) Cost estimation techniques, and (3) Practices and techniques for decision-making. They mapped these elements to the stakeholders' point of view, specifically, for business organizational management, engineering management, and software engineering areas.

Several authors proposed definitions for TD and its interests. Nugroho et al. [4] define TD as *"the cost of repairing quality issues in software systems to achieve an ideal quality level"* and the interests of the debt as *"the extra maintenance cost spent for not achieving the ideal quality level."* Other works try to empirically correlate TD with software size, software quality, customer satisfaction, and other software properties, in the context of enterprise software systems [5].

In a recent Dagstuhl Seminar [6], the following definition of TD was proposed: *"In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability."*

The Software Quality Assessment based on Lifecycle Expectations (SQALE) method [7] addresses a set of external qualities (like Reliability, Efficiency, Maintainability, etc.). Each of these qualities is associated with a set of requirements concerning internal qualities, each provided with a *"remediation function,"* which represents the cost of changing the code so that the requirement is satisfied. Based on these functions, the cost of TD is computed for each external quality and for all qualities.

The Object Management Group has published a beta version of the specification of a measure of TD principal, defined as *"The cost of remediating must-fix problems in production code"* [8]. The measure can be computed automatically as a weighted sum of the *"violations of good architectural and coding practices,"* detected according to the occurrence of specific code patterns. The weight is computed according to the expected remediation effort required for each violation type.

B. Tools Detecting Technical Debt

Several tools are now available to detect TD automatically. Examples of tools are (alphabetically ordered): CAST [9], [10], SonarCloud [11], Squore [12], [13], TeamScale [14], [15], and many more.

The CAST Research Labs (CRL) is part of the enterprise CAST Application Intelligence Platform and is focused on the calculation of the TD for software applications by collecting metrics and structural characteristics of software. CRL also returns insights that can help developers improve the application structural quality.

SonarCloud (aka SonarQube) is an open source cloud platform where software developers can upload their software and collect a set of quality metrics, bugs, vulnerabilities, code smells, code coverage, and code duplications. Starting from all these data, SonarCloud estimates the TD of the software under analysis. Developers may study also the evolution of the TD over time by navigating interactive charts.

Squore is a commercial solution similar to SonarCloud. A set of dashboards visually report quality metrics and 4 key insights (i.e., the overall rating of the software under analysis, trend analysis, forecasts, and project portfolio comparison). Moreover, a specific section is related to TD where four indicators (efficiency, portability, maintainability, and reliability) are used to calculate the TD density and ranking (similar to the six grades used by SonarCloud). The remediation cost shows how many man-days are supposed to be needed to eliminate all TD highlighted.

TeamScale is a commercial tool based on a set of dashboards where managers can monitor the evolution of the quality of their applications. TeamScale is able to analyze the architecture conformance, code clones, missing tests, coding conventions, documentation, external and internal software metrics, and TD. Developers and managers can track the evolution of their software applications by comparing different releases and visualizing the history trend of quality metrics.

All of the mentioned tools can be used to get an overall evaluation of code quality, expressed in terms of TD.

C. System Dynamics

System Dynamics was developed by Jay Forrester [16] as a modeling methodology that uses feedback control systems principles to represent the dynamic behavior of systems. The elements of System Dynamics models are levels, constants, auxiliary variables and rates. The dynamics of systems is determined by how levels work: given a level L, its value in time is always determined by an equation $L(t+\Delta t) = L(t) + (in(t) - out(t))\Delta t$, where *in(t)* and *out(t)* are rates. Levels and rates can concern anything (e.g., people, rabbits, bricks, lines of code, etc.), depending on the application scope and goal of the model. The value of a rate at time t is defined based on the values of auxiliary variables, other rates and levels at time t. Likewise for auxiliary variables, which are not necessary, but are useful to write readable models.

The elements of a System Dynamics model are interconnected just like in the real world, to form a network, where causes and effects are properly represented. Models can be executed, so that the behavior of the modeled system can be simulated. Via System Dynamics models, it is quite easy to perform what-if analyses: you obtain different behaviors by changing the initial state of the system (given by the values of levels), how rates and variable are computed, how they depend on each other, etc.

III. The Proposed Model

As already mentioned, the proposed model describes in an operational way the time-boxed development process, especially in terms of maintenance activities concerning the reduction of TD. The proposed model aims at evaluating the productivity of development and maintenance activities, and the quality of the released product. Productivity is here defined as the ratio of the amount of product—measured in Function Points (FP) [17][18]—developed in a time period to the amount of effort/resources used.

To focus on the main objectives, we abstract from all those aspects of the model that deal with activities and software products that are not directly connected with TD management. For instance, in a real process, the productivity of individuals tends to increase because of learning effects, the number of developers allocated may change during a project, etc.: we exclude all of these variables because they would introduce noise in our investigation, which focuses on the effects of TD management decisions alone.

A. Assumptions

The main reason why practitioners and researchers are interested in TD is that maintaining code burdened with a big TD (i.e., low-quality code) costs much more than maintaining code with little TD (i.e., high-quality code). This is because more work is needed to carry out any code-related activity when code is of low quality (e.g., difficult to understand, poorly structured, full of hidden dependencies, etc.).

To account for the relation that links TD to maintenance cost, we need a measure of TD. To this end, we measure TD via a "TD index," an indicator that takes into account the internal qualities of code that concur to determine the amount of TD embedded in the code. Here, we are not interested in defining precisely the TD index, based on the measures of individual internal qualities, because this is not relevant for our purposes. Clearly, accurately modeling individual internal qualities of code would make the model more apt at reproducing the behavior of real development environments. But this is not our purpose: we aim at building a model that shows—at a fairly high level—the effects of decisions concerning TD management in a generic realistic development environment.

We assume that the TD index ranges between 0 (highest quality) and 1 (worst quality). The extreme values represent limiting cases, which may not occur in practice. When the TD index is 1, maintenance is so difficult that one is better off by simply throwing away the code and building a new version from scratch, and productivity is null, i.e., *prod* = 0. When the TD index is 0, maintenance activities attain their optimal productivity $prod_{opt}$. When 1>TD index>0, *prod* steadily increases from 0 to $prod_{opt}$ when the TD index decreases.

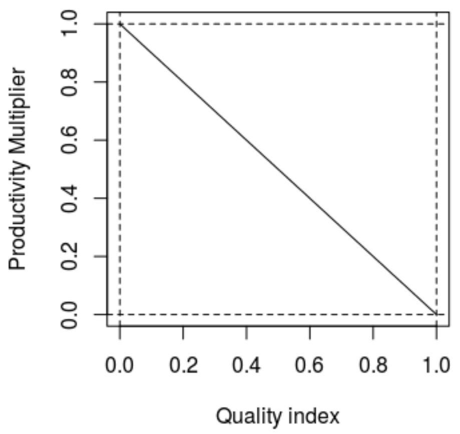

Figure 1. Effect of technical debt on productivity.

The value of productivity for a given value of the TD index *prod(TDindex)* can be expressed as *prodMult(TDindex)* $\times prod_{opt}$. Figure 1 shows a possible behavior of *prodMult(TDindex)*. Namely, Figure 1 implies that there is a linear relationship between TD and productivity: when the TD index is zero (i.e., quality is optimal) productivity is maximum, when the TD index is one (i.e., quality is as bad as possible) productivity is null.

We use the function illustrated in Figure 1 to build models to exemplify our proposal. Other monotonically decreasing functions (e.g., concave or convex function) that go through points $(0,1)$ and $(1,0)$ could be used as well. Here we use the linear model because we have no reason to do otherwise, especially considering that the TD index itself is not thoroughly defined.

Here, we assume that development is carried out in a time-boxed way. This is coherent with the organization of development in most agile processes. We assume that the development is composed of a sequence of "sprints," each of which has a fixed duration and involves a constant number of developers, hence a sprint "consumes" a fixed number of Person Days (PD). For instance, if sprints last 20 work days and involve 5 developers, then each sprint "costs" 100 PD. If at the end of 5 sprints 416 FP are released, we have achieved a productivity of 416/(5·100)=0.832 FP/PD; if at the end of these sprints 378 FP are released, we have achieved a productivity of 378/500=0.756 FP/PD. Quite clearly, in the former case the management of TD was more effective, a higher productivity was achieved, more functionality was released, and bigger returns can be expected.

A consequence of our assumptions is that the amount of effort spent is strictly proportional to development duration, which can be expressed in number of sprints. Given this proportionality between effort and the number of sprints, we can express productivity as the amount of code released after N sprints. Thus, we measure the productivity values above as 416/5=83.2 FP/Sprint (instead of 416/500=0.832 FP/PD) and 378/5=75.6 FP/Sprint (instead of 378/500=0.756 FP/PD).

During each sprint, the developers can carry out two types of activities: 1) increase the functionality of the system, by adding new code, and 2) decrease TD, by refactoring code structure, removing defects and improving the qualities that make development and maintenance easier. Since in each sprint the amount of work is fixed, managers have to decide what fraction of work has to be dedicated to new code development—the remaining fraction being dedicated to TD management. Several different criteria can be used in setting such fraction, as illustrated in Section IV.

We assume that during each sprint a constant fraction of the new code affected by quality problems (hence, increasing the TD) is released. This fraction depends on several factors, like the experience and ability of developers, the availability of sophisticated tools, problem complexity, etc. We assume that these factors are constant throughout all the sprints: in this way, we do not generate noise and we can highlight the effects of TDM decisions.

B. The Model

The proposed System Dynamics model involves two level variables: `CodeSize` (measured in FP) and `TDIndex`.

The constants in the model are:
`nominal_maintenance_productivity`, the productivity in FP/Sprint in ideal conditions, i.e., when the TD index is zero. We assume that the nominal productivity is 80 FP/Sprint, corresponding to 0.1 FP/PersonHour, a fairly typical value [19].
`nominal_TDimprovement_productivity`, the amount of code that can be optimized—i.e., whose TD is completely repaid—in a sprint, when the effort is completely devoted to TD improvement. We assume that this value is 40 FP/Sprint. In real developments, this amount is not necessarily constant: a sprint could be sufficient to "clean" 40 FP or relatively good code, but not to "clean" 40 FP of very bad quality code.
`bad_fraction_of_new_code`, the fraction of the new code (released at the end of each sprint) that contributes to increasing TD. We here assume that the value of this constant is 0.2.
`available_effort`: the effort available at each sprint. As already mentioned, we assume it to be a constant. The actual value is not relevant, however, we can take 100 PD as a reference value.

The rate and auxiliary variables of the model are:
`fraction_of_effort_for_quality_maintenance`: the fraction of `available_effort` dedicated to repaying TD. This variable is computed via function `fracEffortForQuality`, which has the TD index as an argument.
`quality_maintenance_effort`: the effort available for improving the quality of code in a sprint.
`maintenance_effort`: the effort available for developing new code in a sprint.
`maintenance_productivity`: the productivity of developing new code in a sprint. It depends on the `nominal_maintenance_productivity`, the `maintenance_effort` and the decrease of productivity due to the TD (computed via function `productivity_considering_TD`).
`TD_dec_rate`: the TD decrease rate.
`TD_inc_rate`: the TD increase rate.

The values of the aforementioned variables are determined by the following equations:

```
available_effort=1
fraction_of_effort_for_quality_maintenance=
 fracEffortForQuality(TDindex)
quality_maintenance_effort=available_effort*
 fraction_of_effort_for_quality_maintenance
maintenance_effort=
 available_effort-quality_maintenance_effort
maintenance_productivity=
 nominal_maintenance_productivity*
 maintenance_productivity_considering_TD(TDindex)
TDimprovement_productivity=
 nominal_TDimprovement_productivity*
   quality_maintenance_effort
TD_inc_rate=bad_fraction_of_new_code*
 maintenance_productivity/CodeSize
TD_dec_rate=TDimprovement_productivity/CodeSize
```

where the following functions are used:
`maintenance_productivity_considering_TD(TDindex)`: the loss of productivity due to TD, as described in Figure 1.
`fracEffortForQuality(TDindex)`: this function describes the strategy used for tackling TD. In Section IV, we use a few different strategies, hence, a few different function definitions.

The levels are computed as follows (where all auxiliary and rate variables are computed at time t):

```
CodeSize(t+Δt)=CodeSize(t)+
    Δt*maintenance_productivity
TDindex(t+Δt)=TDindex(t)+
    Δt*(TD_inc_rate-TD_dec_rate))
```

IV. Simulating the Model

We simulate the model with a few different TD management strategies. The considered case is characterized as follows. Initially, the software system to be maintained has size 80 FP and its TD index is 0.2 (representing the quality gap between the "ideal" quality and the actual initial quality accepted to speed up development and release the product early). The nominal productivity (i.e., new code development productivity in ideal conditions, when no extra effort is due because of TD) is 80 FP/Sprint. The nominal TD repayment productivity (i.e., the amount of functionality for which the TD is completely repaid in a sprint) is 40 FP/Sprint. At the end of every sprint, 20% of the added code is "bad" code.

Our software organization goes through a sequence of 30 maintenance sprints. We assume that there are always enough new requirements to implement to use up the development capacity of sprints. This is a situation that occurs quite often in practice. We also assume that the same amount of effort is allocated to all sprints. In actual developments, this does not always happen. Anyway, simulations that do not depend on variations in the available effort provide better indications of the effects of TD management strategies, since they do not depend on accidental phenomena, like the amount of available workforce.

A. Constant Effort for TD Management

In the first simulation, we assume that the considered software development organization allocates a constant fraction of the effort available in each sprint, to tackle the TD. It is reasonable to expect that the achieved results depend on how big the fraction of effort dedicated to TD management is. Hence, we run the simulation a few times, with different fractions of the available effort dedicated to TD management, ranging from zero (i.e., nothing is done to decrease the TD) to 40%. The main results of the simulation are given in Figure 2, which shows, from left to right: the functional size of the software product version released after each sprint; the functional size increment due to each sprint (i.e., the enhancement productivity of each sprint); the evolution of the TD through sprints (i.e., the quality of the software product versions released after each sprint).

The amount of functionality delivered at the end of the sprints, and the corresponding TD index are also given in Table I.

TABLE I. RESULTS WITH DIFFERENT FRACTIONS OF EFFORT DEDICATED TO TD MANAGEMENT

Fraction of effort for TD management	Delivered functionality [FP]	Final TD index
0	960	0.77
0.1	1244	0.54
0.2	1461	0.3
0.3	1616	0.05
0.4	1480	0.01

We can examine the achieved results starting with the solid black lines, which represent the case in which no effort at all is dedicated to repaying the TD. It is easy to see that the results obtained by this TD management strategy (a no-management strategy, actually) are quite bad. In fact, after 30 sprints we get only 960 FP: about 500 FP less than the most efficient TD management strategy. Not only: the final product has TD index = 0.77, that is, a very low quality, probably hardly acceptable in practice. The effects of TD on maintenance productivity are apparent: the continuously growing TD makes maintenance less efficient over sprints and after the first 15 sprints, productivity has dropped from 80 FP/Sprint to less than 30 FP/Sprint, due to TD. So, just ignoring the TD is not a good practice. Definitely, we have to allocate some effort to decrease the TD, but how much effort should we dedicate to repaying TD?

By looking at Figure 2 and Table I, it is easy to see that dedicating 10% of the available effort to repaying TD improves the situation with respect to not managing the TD at all: the final size (1244 FP) is bigger, and the final TD index (0.54) is better, though not really good. When we dedicate 20% of the available effort to repaying TD the results improve further: the final size (1461 FP) is bigger, and the final TD index (0.3) is better, though still not very good.

In summary, by increasing the fraction of effort dedicated to repaying TD from 0 to 20% we improve both the amount of functionality that we are able to release, and the quality of the software product. Hence, it would be natural to hypothesize that, by further increasing the fraction of effort dedicated to repaying TD, we obtain improvements in both the amount and quality of delivered software. Actually, this is not the case: when 30% of the available effort is dedicated to repaying TD, we further improve both the released functionality (1616 FP) and the quality (TD index = 0.05), but if an even bigger fraction (40%) of effort is dedicated to repaying TD, we achieve practically ideal quality (the final TD index is 0.01), but substantially less functionality (the final size being 1480 FP).

The explanation of these results is that this is a case of Pareto-optimality: beyond a given point it is not possible to further improve quality without decreasing the amount of released functionality, and vice-versa. Increasing the fraction of effort dedicated to TD improvement clearly improves maintenance productivity by decreasing TD, but at the same time subtracts effort from enhancement maintenance activities. Hence, one should look for a trade-off, to achieve both a reasonably high productivity level and an acceptable quality level (i.e., a sufficiently small TD).

Via a series of simulations, it is possible to find the fraction of effort dedicated to repaying TD that maximizes the released functionality, hence maintenance productivity. In the considered case, allocating 32% of the available effort to TD improvement eventually results in yielding 1638 FP, the final TD index being 0.007.

Finally, it should be noticed that in the short term—i.e., in the first eight sprints or less—not managing TD does not seem to cause relevant negative consequences. For instance, in the considered case, if the goal is to achieve a 400 FP software product, not managing the TD may be a viable choice: you get the product faster than by managing TD. Of course, one should be sure that no further maintenance will be needed, otherwise maintenance cost would be quite high, that is, one has just postponed paying the debt, also adding some interest.

B. Variable Effort for TD Management

In the previous section, the fraction of effort dedicated to quality improvement was fixed, i.e., it was constant over the sprints. This can be used as a first assumption, but it may not be a good managerial choice, for at least the following two reasons. First, the initial TD could be greater than in the case described in Section IV-A. Hence, it would be a good practice to devote a substantial amount of effort to improve quality at the beginning of development, with the objective of decreasing the TD, and then proceed with easier and more productive maintenance. This corresponds to repaying (all or a substantial part of) the TD in the first sprints: the following sprints will have to pay low or null interests.

Second, the effort dedicated to TD management could be excessive. Consider the evolution of the TD index through sprints illustrated in Figure 2: when the fraction of effort dedicated to quality improvement is 40%, the TD is practically nil after 10 sprints. In the following sprints, the fraction of effort for TD management is partly used to balance the increase of debt caused by new code, and part is wasted. This effect is easy to see when you compare the effects of dedicating 30% and 40% of the available effort to TD management. After a few sprints, in both cases the TD index is practically constant (about 0.07 in the former case, about 0.01 in the latter case). Maintenance productivity is also constant in the two cases, but higher in the former case, as shown in the central graph of Figure 2. How is it possible that, when 30% of effort is dedicated to TD management, we are using some effort to manage a higher TD, and still we get a higher productivity?

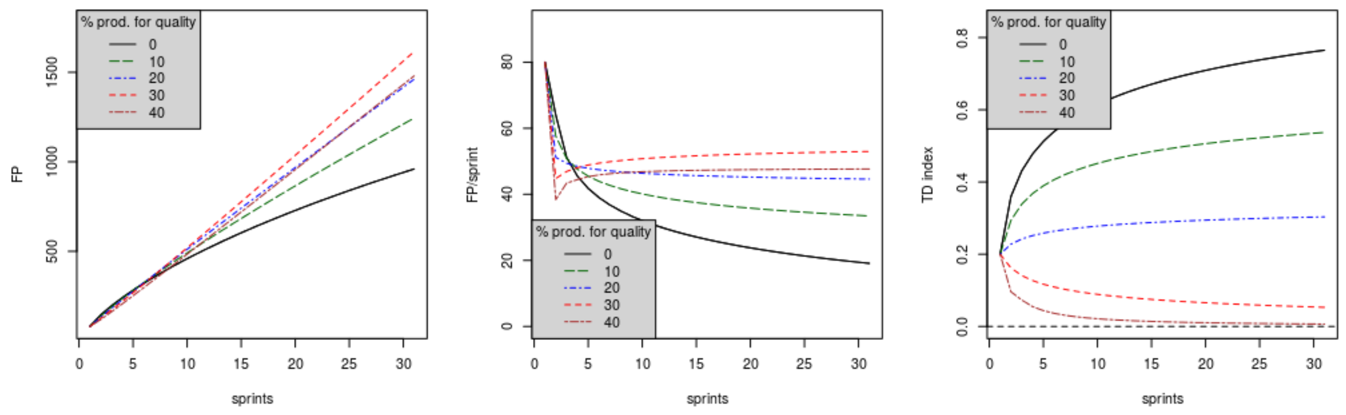

Figure 2. Size of delivered code, Sprint productivity and TD index, depending on a constant fraction of effort allocated to improving the TD.

Because the effort needed to keep TD close to zero is much less than the allocated 40%: the exceeding part is wasted.

A better strategy for TD management would be to allocate to TD improvement a fraction of effort that is larger when the TD is large and smaller when the TD is little. Of course, there are various ways to decide the fraction of effort to be dedicated to decrease TD. We adopt the function shown in Figure 3, which defines the fraction of effort for TD improvement as $1-(1-TDindex)^k$. By changing the value of k we decide how aggressive the approach to debt repayment is: with $k = 1$ the fraction of the effort dedicated to debt repayment is proportional to the debt; with $k > 1$ as soon as TD index raises above zero, a substantial fraction of the effort (the greater k the bigger the fraction) is dedicated to decrease TD.

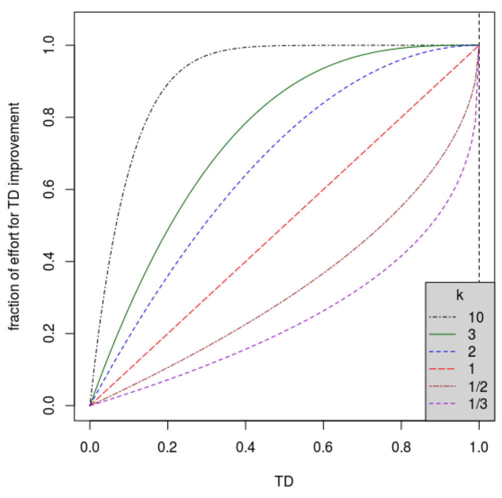

Figure 3. Percentage of effort dedicated to TD improvement, as a function 1-(1-TDindex)^k of TD.

In this section, the fraction of effort dedicated to TD management is decided at every sprint, as $1-(1-TDindex)^3$: a moderately aggressive policy. When debt increases, we try to decrease it fairly soon, to avoid paying large interests. Figure 4 shows the results of the simulation. The adopted policy provides good results: at the end of the sprints we get 1653 FP, more than in any of the simulations performed in Section IV-A. The final TD index is < 0.1, that is, a very good quality.

It is interesting to note that after a few sprints, the TD index remains constant, and, as a consequence, productivity remains constant as well. The reason is that, at the beginning of each sprint, the effort dedicated to TD management is adequate for repaying the existing TD, but, during the sprint, new TD will be created. This situation is perpetuated over the sprints. To completely repay TD, a policy should allocate enough effort to both repay the existing TD, and to *anticipate* the new TD, by performing maintenance in a way that preserves optimal code structure and quality.

However, the strategy simulated in this section dedicates a large fraction of effort to decrease the TD in the first sprints, which guarantees very good results, in terms of both the amount of functionality delivered and the delivered quality.

C. Dedicating Sprints to Technical Debt Removal

In time-boxed development, it is often the case that a sprint is either completely dedicated to enhancement or to decreasing TD (especially via refactoring). So, the policy for allocating effort to TD management is simple: if the TD index is sufficiently high (i.e., above a given threshold), the next sprint will be completely dedicated to TD repayment; otherwise, the next sprint will be dedicated completely to maintenance. In our case, if a sprint is dedicated completely to TD management, developers will be able to optimize a portion of code 40 FP large. Hence, we can allocate a sprint to TD management when a portion of code of at least 40 FP is affected by TD: this is the threshold for deciding to stop developing and have a refactoring sprint.

We simulated development with this criterion for allocating effort to TD management and we obtained the results illustrated in Figure 5. It is easy to see that this strategy results, on average, in two consecutive development sprints followed by a refactoring sprint. TD progressively decreases until it becomes practically nil (oscillating between 0.01 and 0.03). At the end of sprints, 1623 FP are released, that is, a bit less than with the policy described in Section IV-B. However, at the end of refactoring sprints the achieved TD index is better, compared to the TD index achieved in Section IV-B.

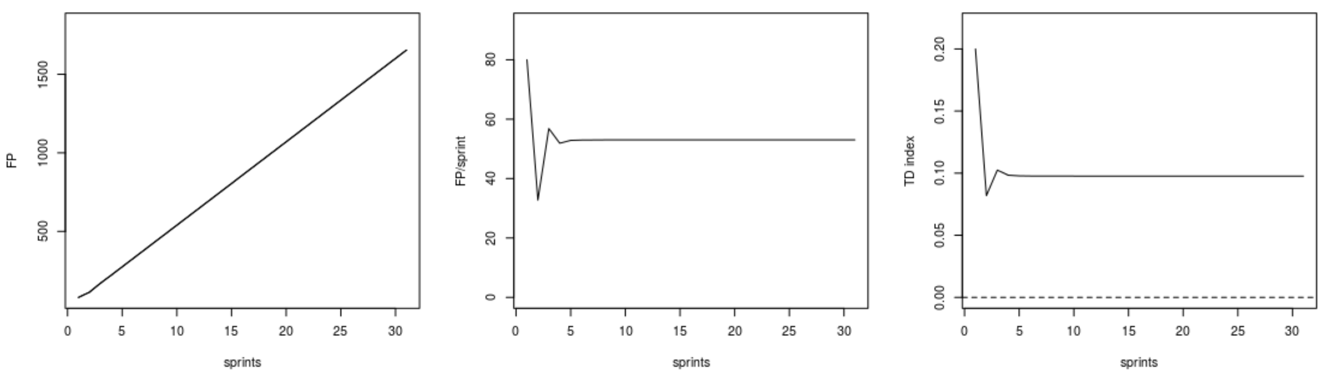

Figure 4. Size of delivered code, Sprint productivity and TD index through sprints, when the fraction of effort for TD improvement is 1-(1-TDindex)^3.

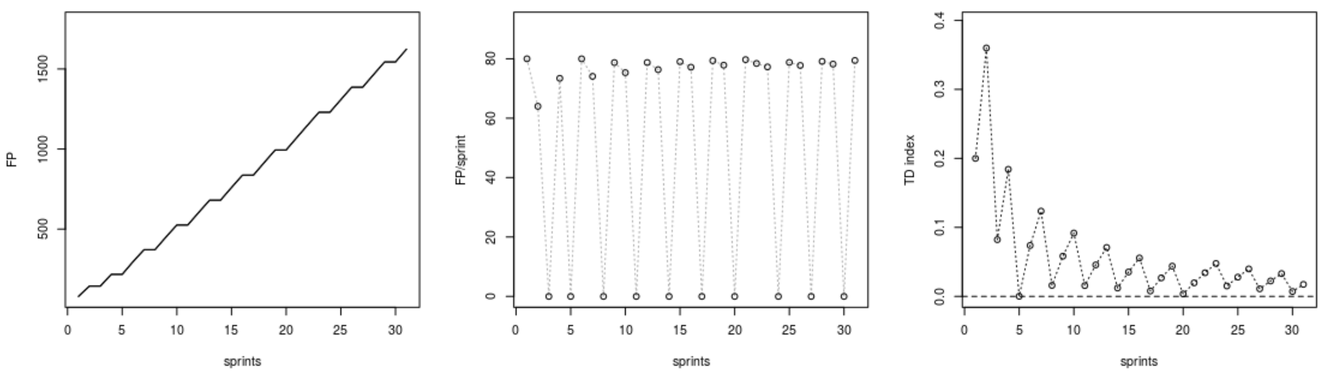

Figure 5. Size of delivered code, Sprint productivity and TD index through sprints, when sprints are dedicated to either TD management or maintenance.

As a final observation, we should consider that applying this strategy is relatively easy, while applying the 'variable fraction' strategy described in Section IV-B is more difficult: with that strategy, both enhancements and refactoring are performed in each sprint: event though this is a fairly natural way of working for developers, it is difficult to assure that exactly the planned amount of effort is dedicated to refactoring.

V. DISCUSSION

The results obtained with the different criteria for allocating effort to TD improvement are summarized in Table II. In Table II, we have added the results—not given in Section IV-B—obtained when the fraction of effort dedicated to TD improvement is 1-(1-TDindex)$^{1/3}$. In such case, the fraction of effort dedicated to TD improvement decided at the beginning of each sprint is based on the current TD index, but the approach is not aggressive. On the contrary, a substantial fraction of effort is dedicated to TD improvement only when the TD index is relatively large.

The results given in Table II, along with the more detailed results reported in Section IV, suggest a few observations.

First, allocating a constant amount of effort to TD improvement does not seem to be a good idea. In fact, if the chosen fraction of effort allocated to TD improvement is too high or too low, the productivity of enhancement maintenance will be lower than possible. Also, the final quality of the product (as indicated by the TD index) could be quite low. In practice, allocating a constant amount of effort to TD improvement works well only if the right fraction of effort is allocated, but choosing such fraction may not be easy.

TABLE II. RESULTS WITH VARIOUS CRITERIA

Criterion	Delivered functionality [FP]	Final TD index
Constant (0%)	960	0.77
Constant (10%)	1244	0.54
Constant (20%)	1461	0.30
Constant (30%)	1616	0.05
Constant (40%)	1480	0.01
1-(1-TDindex)^3	1653	0.10
1-(1-TDindex)$^{1/3}$	1282	0.44
Threshold	1623	0.02

On the contrary, computing the amount of effort for TD improvement at the beginning of each sprint, based on the current TD index seems very effective, especially as far as optimizing the productivity of enhancement maintenance is concerned.

One could observe that in some situations it may be hard to separate clearly the effort devoted to enhancements from the effort devoted to TD improvement. This is particularly true when developers perform refactoring activities while they are enhancing the existing code. For this reason, an organization may want to have sprints entirely dedicated to refactoring and other TD improving activities, and sprints entirely dedicated to enhancements. In this case, the evaluations given in Section IV-C show that allocating an entire sprint to TD improvement whenever there is enough TD to absorb one spring effort provides quite good results, in terms of both productivity and quality.

In any case, we have to stress that all the presented strategies for TD management are based on the quantitative evaluation of TD, which results in the TD index. So, devising a way to measure TD appears fundamental to managing TD effectively and efficiently.

VI. Consequences of a Wrong Evaluation of Technical Debt

Based on the considerations given in Section V above, an organization may decide to adopt the strategy that allocates entire sprints to decrease TD, whenever the TD index becomes big enough, i.e., when the effort spent in a sprint can be absorbed entirely by refactoring and other TD decreasing activities.

Now, the value of the TD index is computed based on the analysis of the code quality performed by tools. Let us suppose that the evaluation reported by tools is not accurate, or that the computation of the TD index based on such evaluation is biased. This will likely result in dedicating too much or too less effort to TD management. In this section, we study the dependence of the effectiveness of managerial decisions concerning TD management on the accuracy of the TD index used to take those decisions.

A. Revised System Dynamics Model

In the model given in Section III-B we have

```
fraction_of_effort_for_quality_maintenance=
 fracEffortForQuality(TDindex)
quality_maintenance_effort=available_effort*
 fraction_of_effort_for_quality_maintenance
```

where the function `fracEffortForQuality(TDindex)` specifies the strategy used for tackling TD, based in the TD index. In this model, variable TDindex is assumed to represent the *real* amount of TD currently associated with the code.

We modified the model as follows:

```
reportedTD=perceivedTD(TDindex)
fraction_of_effort_for_quality_maintenance=
 fracEffortForQuality(reportedTD)
quality_maintenance_effort=available_effort*
 fraction_of_effort_for_quality_maintenance
```

where function `perceivedTD(TDindex)` provides the amount of TD that is perceived by the project manager. Such value depends on the actual `TDindex` and possibly on many other factors. As already mentioned, here we are not interested in defining a detailed model that accounts for all the relevant factors that may affect development, but only to highlight the effect of a few factors specifically related to TD management. Therefore, we define function `perceivedTD(TDindex)` simply as returning $\texttt{TDindex}\times\texttt{TDperceptionFactor}$, where `TDperceptionFactor` is a constant. Clearly, when `TDperceptionFactor` is one, the perceived value of the TD index is the real value of the TD index, while $\texttt{TDperceptionFactor} < 1$ (respectively, > 1) indicates an underestimation (respectively, overestimation) of the TD index.

The value given by function `perceivedTD(TDindex)` is assigned to variable `reportedTD`, which is then used as the argument of function `fracEffortForQuality` to decide how much effort should be dedicated to TD maintenance. With the adopted strategy, if this effort is greater or equal than the effort available in a sprint, the next sprint is dedicated completely to decreasing the TD.

B. Simulating the Effects of Wrong Evaluations of Technical Debt

We simulated the system with different values of `TDperceptionFactor`.

The first outcome we obtained is that when the TD index is moderately overestimated, the results tend to improve, even though only marginally. For instance, when `TDperceptionFactor`=1.2, i.e., when a TD index equal to 0.2 is perceived as 0.24, we get the results described in Figure 6. After 30 sprints, we get 1629 FP and a (real) final TD index slightly above = 0.02. That is, we slightly increased the amount of released functionality at the expenses of very little (practically negligible) decrease of quality.

Underestimating the TD might appear to be more dangerous: when the TD index is underestimated by 10%, we get the results in Figure 7.

After 30 sprints, we get 1588 FP and a (real) final TD index slightly above = 0.03. Although the loss of quality is marginal, we get 46 FP less than when perceiving the amount of TD exactly.

By increasing the underestimation, the results do not change dramatically. When the perceived TD is 75% of the actual TD, we achieve 1546 FP with a still quite good quality (TD index is 0.04).

Probably, we should be more worried about overestimation. In fact, tools performing static code analysis tend to reveal many “violations” of code correctness rules, that may induce project managers—with special reference to those not having a deep understanding of how tools work—to think that their code carries a much larger TD than it actually does.

With our simulated system, a breakpoint occurs when the TD index is overestimated by 135% or more. In such cases, we get the situation depicted in Figure 8. Figure 8 shows that the overestimation of TD induces the project manager to dedicate every second sprint to TD removal. This greatly decreases the amount of effort dedicated to enhancement. The released functionality at the end of the sprints is thus just 1280 FP.

C. Considerations on the Required Accuracy of Technical Debt Evaluation

The simulations described in Section VI-B above show that the correct evaluation if the quantity of TD is important to achieve optimal results. The effects of errors in evaluating the TD index are summarized in Table III.

It can also be observed that the adopted strategy, namely the allocation of entire sprints to TD removal when there is enough TD to justify such allocation, is fairly robust with respect to errors in evaluating the amount of TD. In fact, when the TD evaluation error is relatively small, the ‘pattern’ of enhancement and TD removal sprints remains unchanged. This is evident by looking at the central graphs of Figures 5, 6 and 7: in all these cases, after the initial sprints we have a TD

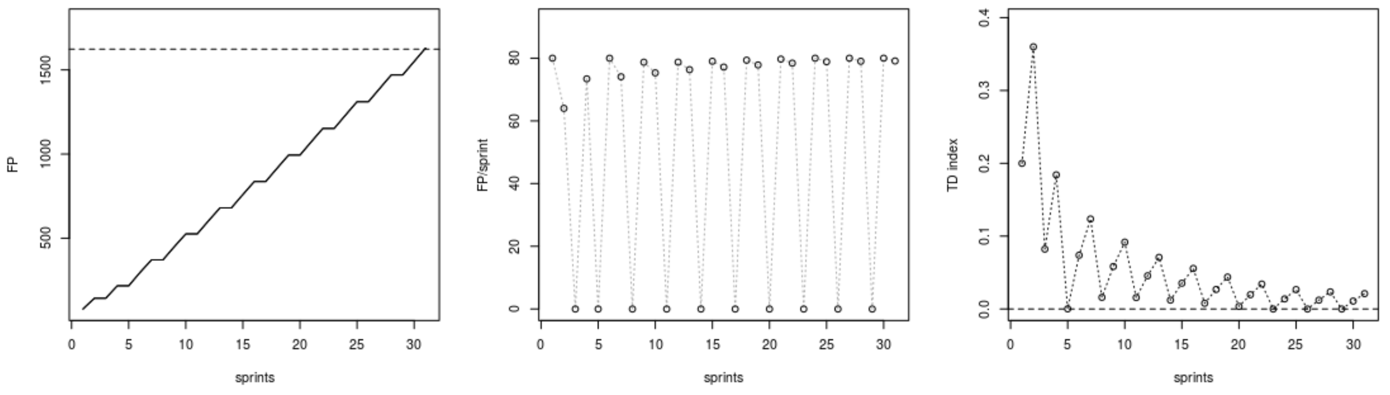

Figure 6. Size of delivered code, Sprint productivity and TD index through sprints, when sprints are dedicated to either TD management or maintenance and TD index is overestimated by 20%.

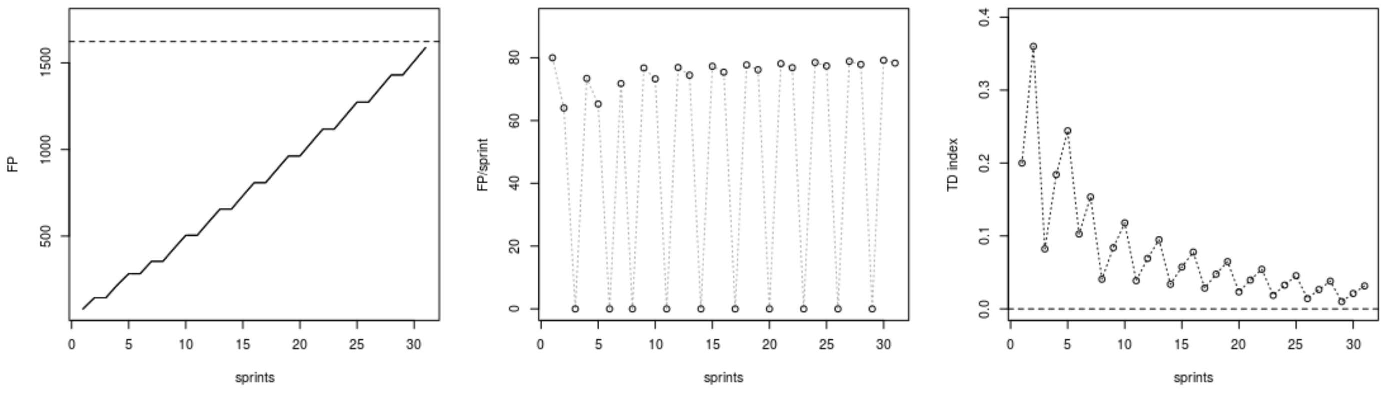

Figure 7. Size of delivered code, Sprint productivity and TD index through sprints, when sprints are dedicated to either TD management or maintenance and TD index is underestimated by 10%.

TABLE III. EFFECTS OF ERRORS IN EVALUATING TD

Error	Delivered functionality [FP]	Final TD index
-25%	1546	0.03–0.04
-10%	1588	0.02–0.03
0%	1623	0.01–0.02
+10%	1626	0.01–0.02
+25%	1629	0.01–0.02
+50%	1556	0.00–0.02
+100%	1565	0.00–0.01
+135%	1257	0.00–0.01

removal sprint every three sprints, with just an exception (in Figures 5 a sequence of three enhancement sprints is present). So, the variations in the amount of delivered functionality are due only to the fact that productivity depends on the amount of TD in the code.

On the contrary, when the overestimation of TD is large, we have that every second sprint is dedicated to TD removal (see the central graph of Figure 6): half of the project development effort is dedicated to removing non-existent TD.

In conclusion, it appears useful to evaluate the TD as correctly as possible, and it is absolutely necessary to avoid large overestimations of TD.

VII. RELATED WORK

The term "technical debt" (TD) was forged by Cunningham [20]. Cunningham observes that sometimes code whose internal quality is not really optimal is released to achieve some immediate advantage, e.g., to ship a product in the shortest possible time, and that *"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. [...] The danger occurs when the debt is not repaid"* because *"excess quantities [of immature code] will make a program unmasterable."*

TD has received a good deal of attention from researchers. For example, a recent Systematic Mapping Study on TD and TD management covering publications from 1992 and 2013 detected the existence of 94 primary studies that were used to obtain a comprehensive understanding on the TD concepts and an overview on the current state of research on TD management [2]. As for TD management, the study reported that some activities—such as TD identification and measurement—received a good deal of attention, while other activities—such as TD representation and documentation—did not receive any attention at all. Finally, a clear lack of tools for managing TD was highlighted: only four tools are available and dedicated to managing TD.

In another Systematic Mapping Study [21], among others,

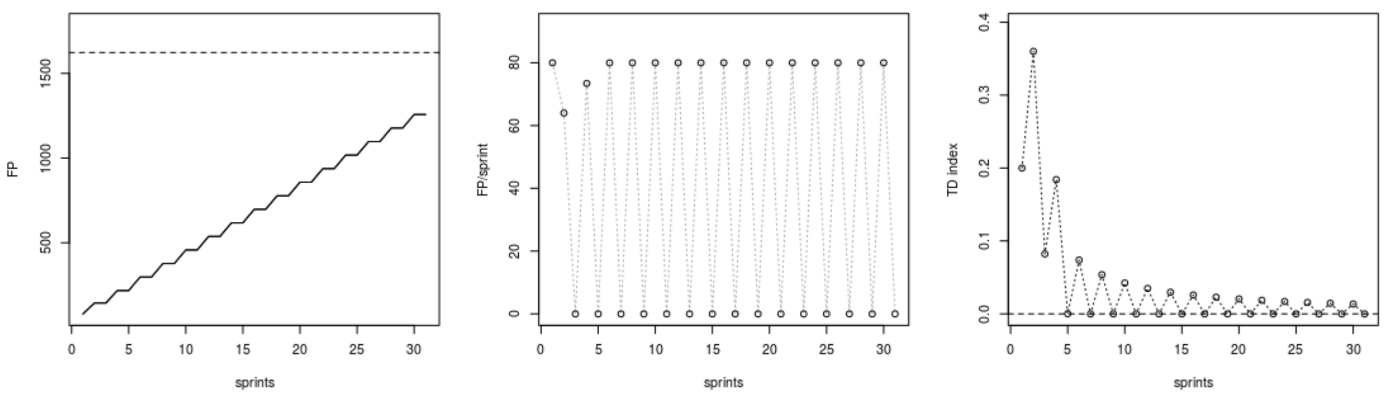

Figure 8. Size of delivered code, Sprint productivity and TD index through sprints, when sprints are dedicated to either TD management or maintenance and TD index is overestimated by 135%.

the following research question was addressed: *What strategies have been proposed for the management of TD, which empirical evaluations have been performed, and which software visualization techniques have been proposed to manage TD?* The findings of the study show that most studies deal with TD at the source code level (i.e. design, defect, code and architecture debt) and researchers detected the existence of TD throughout the entire lifecycle of the project. This implies that ensuring the quality of the project's source code is not the only way to enhance project quality. However, researchers limit the study to the existing problems in the source code. Several studies focus on strategies to manage TD. However, only five strategies (Portfolio Approach, Cost-Benefit Analysis, Analytic Hierarchy Process, Calculation of TD Principal, and Marking of dependencies and Code Issues) were cited and evaluated in more than two papers. Few studies addressed the evolution of TD during the development and maintenance phases of a project.

System Dynamics was first applied in Software Engineering by Abdel-Hamid and Madnick [22], who proposed a model that accounted for human resource management, software development, and planning and control.

The model by Abdel-Hamid and Madnick was used to study software effort and schedule estimation [23], project staffing [24], project control with unreliable information [25], as well as several other aspects of software project management and development.

A software tool for writing and simulating System Dynamics models dealing with software development was also developed [26].

System Dynamics models were also used for simulating, understanding and optimizing the software development process [27] and various activities involved in software development, like requirements engineering [28], reliably control [29], knowledge management [30], outsourcing [31], [32], security [33], and system acquisition [34].

System Dynamics was then extensively used to model software development and its management. A survey of System Dynamics applied to project management was published by Lyneis and Ford [35], while in [36] De Franca and Travassos propose a set of reporting guidelines for simulation-based studies with dynamic models in the context of SE to highlight the information a study of this type should report.

Cao et al. [37] proposed a System Dynamics simulation model that considers the complex interdependencies among the variety of practices used in Agile development. The model can be used to evaluate—among others—the effect of refactoring on the cost of implementing changes. The model proposed by Cao et al. is quite comprehensive: it includes sub-models of human resource management, Agile planning and control and Customer involvement, which are not present in our model. Besides, the models of Software production, Change management, and Refactoring and quality of design are more detailed than in our model. In fact, the model by Cao et al. aims at providing a complete and realistic simulation of real Agile processes, so it needs to be complete and detailed, while we just aim at showing the occurrence of phenomena that are described by the literature on TD.

Glaiel et al. [38] used System Dynamics to build the Agile Project Dynamics model, which captures each of the Agile main characteristics as a separate component of the model and allows experimentation with combinations of practices and management policies. Like our model, the APD model addresses developments that proceed through sprints and includes the representation of TD; nonetheless, the APD model is quite different from ours. In fact, the APD model is quite complex, as it aims at capturing several (if not all) of the characteristics of agile development, while our model is a proof of concept, aiming at showing how the TD works. As a result, the effects of basic managerial decision (like the proportion of effort to be devoted to quality improvement) on TD are more evident in our model.

We remind the reader that our model is by no means suitable for realistic simulations, being oversimplified. The ADP model is for sure more suitable for simulating the likely behavior of a real agile development project.

Finally, it should be noted that both Glaiel et al. [38] and Cao et al. [37] model agile software development processes, while we propose a model of a software development process that—although incremental and time-boxed—is not constrained by several features of typical agile development processes.

A consequence of this difference is that the models by

Cao et al. and by Glaiel et al. consider quality improvements exclusively connected with refactoring, while in our model quality improvement is a specific goal, which is supported by ad-hoc budget (the fraction of effort dedicated to quality improvement) and can be performed continuously and in conjunction with development activities.

Cao et al. develop a system dynamics simulation model that –although comprehensive– does not focus on technical debt and how to optimize its management, as we propose to do. Glaiel et al. model technical debt and its management as part part of the refactoring section of a larger model. As such, the effect and the management options for dealing with technical debt are somewhat hidden in the model. On the contrary, our model lets us focus on the technical debt, its effects and the consequences of specific management strategies.

In conclusion, to the best of our knowledge, no other paper addresses the TD management problem as we did. Although less comprehensive than the mentioned System Dynamics models, our proposal helps better understand the consequences of TD and the effectiveness of its management strategies.

VIII. Conclusions

The term "technical debt" indicates several concepts and issues related to software development and maintenance. The latter are complex and multifaceted activities: accordingly, it is not surprising that managing TD is quite difficult [6].

In this paper, we have provided a formal, executable model of time-boxed software development, where the effects of TD are explicitly and quantitatively represented and accounted for. The model is usable to show—via simulation—the effects that TD have on relevant issues such as productivity and quality, depending on how TD is managed, with special reference on how much effort is dedicated to TD repayment and when—in a sequence of sprints—such effort is allocated.

The model also accounts for errors in the evaluation of the amount of TD affecting the code. Measuring the TD is necessary, because the decisions of how much effort should be conveniently subtracted from code enhancement and dedicated to debt reduction is often based on the knowledge of how much debt has been accumulated. In fact, the assumption that more TD decreases productivity leads to increasing the effort for debt reduction, in order to restore high productivity levels. The proposed mode shows how to evaluate –via simulation– the effects of errors in the measure of TD.

The proposed model can be used to prove or disprove concepts and hypotheses, to perform what-if analyses, etc. However, our model is not intended to be used in practical software project management as-is, because, the model illustrated above is too abstract and contains hypotheses that could not match the target development environment. Whoever wants to use the presented model for practical project management should first enhance it; examples of models representing all the main aspects of software development can be found in the papers by Cao et al. [37] and Glaiel et al. [38].

We plan to extend the presented model in several directions: to account for different effects of TD on productivity (i.e., with functions different from the one in Figure 1), to explicitly model defects, to test different debt management policies, etc.

Acknowledgment

This work was partly supported by the "Fondo di ricerca d'Ateneo" funded by the Università degli Studi dell'Insubria.

References

[1] L. Lavazza, S. Morasca, and D. Tosi, "A method to optimize technical debt management in timed-boxed processes," in ICSEA 2018 - The 13th International Conference on Software Engineering Advances, 2018, pp. 45–51.

[2] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," Journal of Systems and Software, vol. 101, 2015, pp. 193–220.

[3] C. Fernández-Sánchez, J. Garbajosa, A. Yagüe, and J. Perez, "Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study," Journal of Systems and Software, vol. 124, 2017, pp. 22 – 38.

[4] 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. ACM, 2011, pp. 1–8.

[5] N. Ramasubbu and C. F. Kemerer, "Managing technical debt in enterprise software packages," IEEE Transactions on Software Engineering, vol. 40, no. 8, Aug. 2014, pp. 758–772.

[6] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)," in Dagstuhl Reports, vol. 6, no. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016, pp. 110–138.

[7] J. Letouzey, "The SQALE method for evaluating technical debt," in Proceedings of the Third International Workshop on Managing Technical Debt, MTD 2012, Zurich, June 5, 2012, 2012, pp. 31–36.

[8] "Automated technical debt measure – beta," Object Management Group, specification ptc/2017-09-08, September 2017.

[9] "CAST Application Intelligence Platform," https://www.castsoftware.com/products/application-intelligence-platform, accessed: 2019-02-05.

[10] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the size, cost, and types of technical debt," in Proceedings of the Third International Workshop on Managing Technical Debt. IEEE Press, 2012, pp. 49–53.

[11] "sonarcloud," https://sonarcloud.io/about, accessed: 2019-02-05.

[12] "squoring," https://www.squoring.com/en/, accessed: 2019-02-05.

[13] B. Baldassari, "Squore: a new approach to software project assessment." in International Conference on Software & Systems Engineering and their Applications, 2013.

[14] "https://www.cqse.eu/en/," https://www.cqse.eu/en/, accessed: 2019-02-05.

[15] V. Bauer, L. Heinemann, B. Hummel, E. Juergens, and M. Conradt, "A framework for incremental quality analysis of large software systems," in Software Maintenance (ICSM), 2012 28th IEEE International Conference on. IEEE, 2012, pp. 537–546.

[16] J. Forrester, Industrial Dynamics. Cambridge: MIT Press, 1961.

[17] A. J. Albrecht, "Function points as a measure of productivity," in Act as do 53 rd meeting of GUIDE International Corp., Guidance for users of integrated data processing equipment conference, Dallas, 1981.

[18] IFPUG, "Function point counting practices manual, release 4.2," IFPUG, Tech. Rep., 2004.

[19] L. Lavazza, S. Morasca, and D. Tosi, "An empirical study on the factors affecting software development productivity," e-Informatica Software Engineering Journal, vol. 12, no. 1, 2018, pp. 27–49.

[20] W. Cunningham, "The wycash portfolio management system," ACM SIGPLAN OOPS Messenger, vol. 4, no. 2, 1993, pp. 29–30.

[21] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," Information and Software Technology, vol. 70, 2016, pp. 100–121.

[22] T. Abdel-Hamid and S. E. Madnick, Software project dynamics: an integrated approach. Prentice-Hall, Inc., 1991.

[23] T. K. Abdel-Hamid, "Adapting, correcting, and perfecting software estimates: a maintenance metaphor," Computer, vol. 26, no. 3, 1993, pp. 20–29.

[24] T. K. Abdel-Hamid, K. Sengupta, and M. Hardebeck, "The effect of reward structures on allocating shared staff resources among interdependent software projects: An experimental investigation," IEEE Transactions on Engineering Management, vol. 41, no. 2, 1994, pp. 115–125.

[25] K. Sengupta and T. K. Abdel-Hamid, "The impact of unreliable information on the management of software projects: a dynamic decision perspective," IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, vol. 26, no. 2, 1996, pp. 177–189.

[26] A. Barbieri, A. Fuggetta, L. Lavazza, and M. Tagliavini, "Dynaman: a tool to improve software process management through dynamic simulation," in Computer-Aided Software Engineering. Proceedings., Fifth International Workshop on. IEEE, 1992, pp. 166–175.

[27] R. J. Madachy, Software process dynamics. John Wiley & Sons, 2007.

[28] F. Stallinger and P. Grünbacher, "System dynamics modelling and simulation of collaborative requirements engineering," Journal of Systems and Software, vol. 59, no. 3, 2001, pp. 311–321.

[29] I. Rus, J. Collofello, and P. Lakey, "Software process simulation for reliability management," Journal of Systems and Software, vol. 46, no. 2, 1999, pp. 173–182.

[30] P. Peters and M. Jarke, "Simulating the impact of information flows in networked organizations," ICIS 1996 Proceedings, 1996, p. 30.

[31] S. T. Roehling, J. S. Collofello, B. G. Hermann, and D. E. Smith-Daniels, "System dynamics modeling applied to software outsourcing decision support," Software process: improvement and practice, vol. 5, no. 2-3, 2000, pp. 169–182.

[32] A. Dutta and R. Roy, "Offshore outsourcing: A dynamic causal model of counteracting forces," Journal of Management Information Systems, vol. 22, no. 2, 2005, pp. 15–35.

[33] ——, "Dynamics of organizational information security," System Dynamics Review, vol. 24, no. 3, 2008, pp. 349–375.

[34] S. J. Choi and W. Scacchi, "Modeling and simulating software acquisition process architectures," Journal of Systems and Software, vol. 59, no. 3, 2001, pp. 343–354.

[35] J. M. Lyneis and D. N. Ford, "System dynamics applied to project management: a survey, assessment, and directions for future research," System Dynamics Review, vol. 23, no. 2-3, 2007, pp. 157–189.

[36] B. B. França and G. H. Travassos, "Experimentation with dynamic simulation models in software engineering: Planning and reporting guidelines," Empirical Software Engineering, vol. 21, no. 3, June 2016, pp. 1302–1345.

[37] L. Cao, B. Ramesh, and T. Abdel-Hamid, "Modeling dynamics in agile software development," ACM Transactions on Management Information Systems (TMIS), vol. 1, no. 1, 2010, pp. 5:1–5:27.

[38] F. S. Glaiel, A. Moulton, and S. E. Madnick, "Agile project dynamics: A system dynamics investigation of agile software development methods," 2014.