Full citation: Holvitie, J., Licorish, S.A., & Leppänen, V. (2016) Modelling Propagation of Technical Debt, in Proceedings of the 42nd Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA). Limassol, Cyprus, IEEE Press, TBC.

Modelling Propagation of Technical Debt

Johannes Holvitie*†, Sherlock A. Licorish‡, and Ville Leppänen*†
* TUCS – Turku Centre for Computer Science,
Software Development Laboratory
† University of Turku,
Department of Information Technology,
Turku, Finland
{jjholv, ville.leppanen}@utu.fi
‡ University of Otago,
Department of Information Science,
Dunedin, Otago, New Zealand
sherlock.licorish@otago.ac.nz

***Abstract*—Facing various constraints, software developers are forced to make trade-off decisions during the software development process. Technical debt management aims to acknowledge and govern these trade-off decisions, the debt's payoff, in the process optimizing the efficiency and sustainability of the project for the long-run. Core to successful technical debt management is delivering up-to-date and accurate information to correct project stakeholders, so that technical debt may be properly accounted for in the project's decision making. Whilst solutions exist for identifying technical debt there are very few solutions that maintain accumulated technical debt information. Noting the overwhelming speed during software development, and particularly in environments where rapid delivery is the norm, the lack of accumulated technical debt information could result in ineffective management. We introduce technical debt propagation channels in this paper to advance software maintenance research on two accounts: (1) We describe the fundamental components for the channels, allowing identification of distinct channels, and (2) we describe a procedure to identify and abstract technical debt channels in order to produce technical debt propagation models. Our propagation models pursue automation of technical debt information maintenance with program analysis results, and translation of the maintained information between existing–and currently disconnected–technical debt management solutions. We expect the immediate technical debt information to enhance applicability and effectiveness of existing technical debt management approaches.**

I. Introduction

A software development method describes a set of practices, processes, and roles that are suitable for producing software in a particular environment. Common environment characteristics include requirement-change-volatility, resource-availability-confidence, and level of regulation. From the early 2000's a number of software development methods were introduced that targeted environments with high requirement change rates and scarce resources [1]. The commonality of these methods is the application of iterative and incremental development. These approaches define cycles of development (i.e. iterations) during which no requirement changes are accepted, and the end result of a cycle is a functional addition (i.e. increment) to the software product [2]. The iterations remove volatility for a short period of time while the increment allows the client to base his/her feedback on the actual product.

In the wake of iterative and incremental software development methods (many of which are referred to as the agile or the lean methods) becoming more frequent, technical debt management has arisen to address the issue of maintaining efficiency [3]. This is particularly necessary, as successive software iterations depend on the completeness and quality of prior iterations in order to deliver new functionality with set resources. However, inoptimalities do emerge due to trade-offs, oversight, or environmental changes, and they persistently affect future iterations until seen to [3]. Technical debt management pursues introducing structure and order into these, often ambiguous, inoptimalities so as to resolve them adequately to the software development project [4], [5], [6].

Technical debt management consists of three phases [7]: identification, estimation, and decisions making. Identification captures technical debt instances, and estimation produces effort estimates for the instances based on upcoming changes and historical data. Knowledge regarding the propagation of existing technical debt within the project is core to this phase's success [8]. The decision making phase applies suitable evaluation process in creating estimates to determine handling for technical debt instances.

It can be noted that technical debt identification is context-dependent as an inoptimality consists of deviations in implemented artifacts, and, hence, the implementation technology defines the context (e.g. the Python programming language or UML for design descriptions). Similarly, estimation requires context-dependent information in order to properly evaluate how technical debt has and will propagate in these contexts, and thus, accumulate or dissipate effort [9].

While the decision making phase, arguably, does not require technology context related information, the information need it imposes on to the estimation phase is project-context dependent. Notably, for current software development methods and practices, the project's required frequency and diversity of information can be notable. For example, the Scrum method [2] sees decision making events both daily and every few weeks in the form of daily stand-ups and Sprint reviews and retrospectives. Concluding, the estimation phase's ability to serve technical debt information as frequently and as appropriately as possible enables technical debt related decision making, even at the level of a single developer making multiple decisions a day regarding work carried out.

Prior research on technical debt has successfully introduced technical debt identification, estimation, and decision making approaches (e.g. [6], [10], [11], [12]), or described how

solutions from other domains can be adopted for these phases (e.g. [8], [13]). However, the majority of the solutions come with preset technology or project contexts which is problematic. Indeed, Holvitie et al. [14] have noted that technical debt is capable of propagating between components that exist in different phases of the software development life-cycle. That is to say, the same technical debt instance may affect design and implementation phases of the software development. Based on this they have further postulated that technical debt is capable of leaving its original technology context [15]. Since both the identification and estimation phases are context dependent, research on how technical debt propagates within and between these contexts is required, but currently absent.

Hence, in this paper we make the proposal for technical debt propagation models, which are abstractions from technical debt propagation channels observed during software development undertakings. The models contribute to technical debt management by explaining how technical debt information transforms from one context to another. This insight paves way for the translation of information between technical debt management solutions and identifies gaps in the current information flow. For instance, our models can be seen to link source code and documentation elements so that technical debt identified in one context could be assessed while taking into account the effort associated with elements in the other (e.g. linking God Class implementation debt solutions [8], [16], [17] to maintenance debt solutions [18], [19]).

Furthermore, given that software development can grow in complexity over time, the models' capability of being programmatically assessed should be highlighted, as manual maintenance is not feasible for busy developers. Qualitative analysis is required to identify technical debt [8], but as long as the identified elements remain static the contexts' available semantics can be used to maintain this information. Hence, tools that allow documenting instances of technical debt could (and should) be superimposed with this maintenance feature (e.g. [20] provides a platform for combining these features).

The rest of the paper is constructed as follows. Section II reviews related work on technical debt propagation and estimation, and software entity interconnections. Section III describes the characteristics of technical debt propagation as channel properties. Section IV defines a method for identifying and abstracting the technical debt channels to produce technical debt propagation models. Discussion on the implications and threats is given in Section V. Section VI concludes the paper.

II. Related Work

To provide a basis for our research, we review existing work on technical debt propagation and estimation in Section II-A, and on software entity relations modelling in Section II-B.

A. Technical Debt Propagation and its Estimation

McGregor et al. [21] hypothesized that there are two ways for technical debt to propagate within ecosystems. Firstly, they noted that the technical debt for a newly created asset is "the sum of technical debt incurred by the decisions made during the asset's development and some of the technical debt from the assets that were integrated to it". Regarding this, they also noted that multiple implementation layers can diminish debt. Secondly, they established that the user of an asset did not accumulate technical debt directly, but the effects of the technical debt were felt indirectly by the user as implications. Regarding both assessments, they note that the compound debt may become larger than the sum of its sources [21].

Along similar liners, as part of his work, Schmid [22] provides a formal definition for technical debt accumulation. Here, an evolution step is defined as an externally observable behavior change that introduces a characteristic to a system. This includes both addition of new functionality as well as quality modifications. Technical debt accumulation (which we interpret as the cumulative effect of technical debt propagation) is described as the difference in costs to implement a sequence of evolutionary measures in the current system, in comparison to an optimal system; where both systems are considered behaviorally equivalent.

Regarding, especially value, estimation of identified technical debt, Zazworka et al. [23] note from their case-study that principal and interest characteristics of technical debt are not bound to the type of technical debt. Eisenberg [24] notes that threshold based management approaches require defining the cost associated with reducing each type of technical debt. Falessi et al. [25] collect requirements for technical debt tool support. For valuation of the debt's interest, they note that a single debt may affect diverse quality characteristics differently. Falessi et al. work also acknowledges the compound property as discussed previously by McGregor et al. [21]. Finally, Zazworka et al. [16] consider refactoring cost and impact on quality characteristics as the two prioritization dimensions for managing specific types of design debt.

From this short literature review we note that previous work has described technical debt propagation capabilities, even exhaustively, but what is lacking is a method for capturing the different ways and forms with which technical debt can propagate within software projects. Noting that the reviewed work has also argued for the importance of acknowledging differences in the value generation of different types of technical debt, the authors believe that it is timely to develop technical debt propagation models. In fact, these would contribute greatly towards the realization of managing technical debt propagation capabilities for particular settings.

B. Software Entity Interconnections

Kim et al. [26] discuss an approach for classifying software changes. In their approach, they first extracting change history for projects from the projects' software configuration management systems. From this history, the bug-introducing changes are identified and feature extraction is applied for them in order to produce a classifier. Notably, bug-introducing changes are identified by back-tracking from the bug-fixing change. Further, feature extraction takes into account not only the program implementation but also the associated log messages [26].

The Software Process Engineering Metamodel (SPEM) [27] can be seen to extend the inter-connectivity of software entities beyond the software implementation. The SPEM pursues formalization of software processes via definition of their process components, component relations, and the impulses flowing within. Effects of the interconnections are not described by the model, however the work of Rochd et al. [28] can be seen explaining this via superimposing synchronization for the modeled components. Synchronization events are all modifications that affect the process structure or the process contents.

III. Technical Debt Propagation Channels

This section provides the theoretical rationale for capturing technical debt propagation as channel descriptions. Facilitation is given as a definition of technical debt channels' common features in Section III-A and information properties in Section III-B.

A. Channel Features

Our objective is to describe technical debt propagation channels capturing the effects of inoptimal software entity alterations, or their possible implications on the hosting organization (implications are discussed as realization probabilities in Section III-B3). Software entity alterations correspond to software changes as argued by Holvitie et al. [15]. These changes (entity alterations) are captured here as Entity-Relationships (ER) aligned with Kagdi et al.'s [29] definition of software change as "the addition, deletion, or modification of any software artifact such that it alters, or requires amendment of, the original assumption of the subject system".

As a technical debt channel captures a particular, distinct, instance of technical debt accumulation, the channel's definition is always comprised of a single entity-entity-relationship. Their combination would correspond to an instance of a synchronized SPEM (describing the technical debt propagation process for—i.e. the channels available to—instances of technical debt in the project's specific set of contexts).

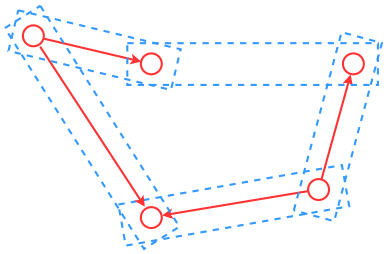

Fig. 1. Software entities and their relationships (red) and technical debt channels (blue; dotted lines)

Figure 1 demonstrates a collection of software entities (e.g. variable and method declarations and calls in the implementation technology's context and object's describing these in the documentation technology's context) and their relationships. Highlighted on top of this dependency graph (red lines) are the potential channels for software change (dotted lines); a super-set of the dependency graph. The super-set assumption holds when one notes that the definition for a software change also considers assumptions. Section III-A1 describes these assumptions as implicit channels.

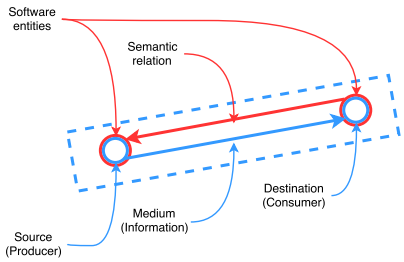

Fig. 2. Software entity relationship (red; top descriptions) with a super imposed technical debt channel (blue; bottom descriptions); the semantic relation is not a requirement for the channels existence

Figure 2 depicts one instance of a potential software change (from Figure 1). As per the previous description of a software change between software entities, this directed relationship corresponds to a technical debt channel. From the viewpoint of technical debt accumulation, the entity which invokes a potential software change is the *source* of technical debt (entity on the left), the relation which delivers the invocation corresponds to a channel *medium*, and the entity in which the potential change will take place is the *destination* of technical debt. The following sections will further define these parts.

1) Mediums: A medium is "a system with the capability of effecting or conveying something" [30]. Herein, a medium is described in the technical debt context through the information that is carried and through the system capable of conveying the information. The information that is carried (1) *describes changes within the source*, and (2) *indicates changes in the destination* (or implies them as per Section III-B3).

The system that is capable of carrying this information is context dependent, and Suovuo et al. [31] have argued that the system is either explicit or implicit (see Figure 3). An explicit system relies on pre-existing context semantics which define operations for other systems (e.g. dependency invocation or inheritance for common implementation contexts, and UML notation conventions or standardized logging structures for documentation contexts). Note that channel direction is generally opposite to dependencies.

Implicit channels do not have a formal counterpart and may thus expand to areas that formality disallows, especially in relation to unions of software contexts, such as developer's conceptualization that takes place between design documentation and component implementation. Due to their unobtrusive nature, implicit channels are difficult to observe [31].

2) Sources and Destinations: The sources and destinations of a technical debt channel capture the *information producers* and *consumers* of the medium respectively. A source is an entity that exists in a context. It produces information regarding changes in the entity. The information regarding the change must be observable from outside the entity in order for the information to ever reach the medium. Hence, for

a source entity, a valid type is a declaration type that can be referred. Thus, the source entity types correspond to the hosting software entity's context's referable type definitions.

Similarly, the destination exists in a context and is capable of receiving and consuming information regarding the source entity by way of being connected to it through a medium. Hence, valid channel destination entity types are the software entity's context's definitions capable of making references.

The source and destination entities can exist in different contexts (a feature noted especially for implicit channels). The source entity can not be the destination entity. As in these scenarios, the information is consumed where produced with no outside observable effects, and this deviates from the definition provided for software change(s) (c.f. [29]).

B. Information Properties

Following the definition of a medium [30], a channel interacts with the environment in which it exists. Section III-A1 established the general process for this as the two capabilities of the delivered information: (1) description of changes within the source, and (2) indication of changes in the destination (or, in certain cases, implication of changes). This section describes how the channel's interaction adheres to the technical debt management environment by describing properties of the delivered software change information.

Technical debt research has reached adequate consensus regarding the properties that an instance of technical debt has [7], [13], [16], [32], [33]: a location, a principal, an interest, and an interest realization probability. A technical debt propagation channel conveys information that realizes technical debt instances for an evolving software project (i.e. the instance accumulates or dissipates value as its propagation path expands or contracts via the available technical debt propagation channels [22]). From this perspective, the instance's location property is directly related to the entities forming the sources and the destinations of the technical debt channels which form the instance's propagation path. The rest of the properties are related to the details in the following three subsections.

1) Principal: A technical debt instance captures the increase in effort caused by inoptimalities in a particular location within a software development project. The increase corresponds to the effort difference between retaining the location's current state and making adjustments to achieve its optimal state. The principal is the portion from the effort increase that corresponds to bringing the root cause (initial accumulation point for the difference) to optimum [7], [13]. Schmid's formalization [22] (see Section II) explains that technical debt is accumulated when software evolution consumes more resources than the optimal evolution would. Hence, the information carried by the technical debt channel accumulates principal, for the instance, in this entity if 1) *the software change indicates additional resource consumption* and 2) *the entity hosts the technical debt instance's root cause*.

2) Interest: The interest of a technical debt instance captures the extra resources that are spent due to the principal's existence, but in entities that do not host the principal [3]. Thus, the definition for the channel information that accumulates interest in entities is analogous to the one given for principal (refer to Section III-B1). However, the second condition is inverted: the information carried by the technical debt channel accumulates interest, for the instance, in this entity if (1) *the software change indicates extra resource consumption*, and (2) *the entity does not host the root cause of the technical debt instance*.

3) Realization Probability: The realization probability of a technical debt instance corresponds to the chance that further resource consumption is initiated by the instance's existing debt. This can be seen to take place either in the principal or the interest locations, or in locations connected to them [21]. From the perspective of capturing technical debt propagation channels, the realization probability becomes a special measure of an entity-entity-relationship's existence. In this, the source entity hosts technical debt from the instance, the destination is an entity wherein currently observed resource consumption has not yet taken place, and the relationship describes a relation between these entities. The realization probability measure indicates the chance of this system becoming a technical debt propagation channel.

By the definition of principal and interest information, when the observed realization probability is lower than one (i.e. certainty) the channel is not a technical debt propagation channel as no technical debt information is delivered yet. Rather, it is a potential future technical debt channel. Project management can, however, acknowledge these potential channels and strategize to account for their impact prior to realization.

IV. Technical Debt Modelling

This section describes the method for locating technical debt channels, classifying them, and abstracting the classes into technical debt propagation models. Section IV-A describes the tripartite process for this method, while Section IV-B provides initial validation in assessing its usefulness.

A. Process

Technical debt channels describe systems for accumulating technical debt. Operationalizing such a system should hence dissipate technical debt: as a system evolves, an instance of technical debt propagates through technical debt channels forming a propagation path for the instance. The software development life-cycle contains multiple implementations of these systems (e.g. refactoring, rearchitecting, review, and so forth) that produce historical data. This can be used in order to identify technical debt dissipation, and thus, be inverted in order to produce technical debt channels.

1) Fixing the Observation Level: Technical debt describes impaired software evolution for which technical debt channels describe the system responsible for this evolution in single software entities (see Sections II and III-A). Fixing the observation level of the historical software change information is a prerequisite for identifying the channels, as we must pinpoint, for each software entity, the specific pieces of change information that describe evolution solely for this

entity. Formally, the observation level must provide such time and partition granularity for the change information that it allows identifying each software entity's $e \in E$ evolution as a sequence of states $e : (s_1, s_2, ..., s_n)$.

2) Identifying Technical Debt Channels: Observing technical debt instances' propagation, from historical data, corresponds to identifying cause-and-effect relations for the software changes observed for the entities [15]. These relations are captured for the entities' state sequences (see Section IV-A1) as identified pairs

$$r = ((e_1, s_i), (e_2, s_j)) \in R \mid (e_1, s_i) \rightarrow (e_2, s_j) \qquad (1)$$

Pair r indicates that entity e_1's state s_i has caused a state to change to s_j in another entity e_2. Further, let $d(e, s)$ be the time stamp that relates to entity e's state s, and $D_{e_1,d_0} = \{s \mid d(e_1, s) \geq d_0\}$ be entity e_1's group of states for which the time stamp is greater than or equal to d_0. Hence, the prerequisite for pair r's causality in Eq. 1 is that $s_j \in D_{e_2,d(e_1,s_i)}$.

As Section III-A2 describes the source and destination entities of a technical debt channel as the information producer and consumer respectively, we find the components of a technical debt channel capturing r as follows. The channel's source entity e_s produces the information in $r = ((e_1, s_i), (e_2, s_j))$. Hence, from Eq. 1 we get $e_s = e_1$. Analogously for the destination, $e_d = e_2$. Last, the channel's medium is described (see Section III-A1) via the carried information and thus corresponds to the information realizing $(e_s, s_i) \rightarrow (e_d, s_j)$.

Example of the previous is identification of a technical debt instance's removal through observing a series of consecutive version control entries wherein comments, tags, or other meta-information can be used to associate the entries' contents to the instance. The comments justify associating changes to the instance's removal, while the chronological ordering of the entries allows forming cause-and-effect relations for the changed entities. An entity which describes a cause for a change corresponds to a technical debt information producer, while an entity for which an effect is observed corresponds to the information's consumer. Also, only one producer and consumer should be found for a single piece of information. If multiples are found, the cause-effect-relationships between entities are not evident, and the observation level must be lowered by way of decomposing the information further [15].

It should also be noted that Section III-B describes the properties for information that corresponds to technical debt propagation. In associating the information to entities, presence of these properties should be ensured in order to only capture technical debt propagation channels, and not software change propagation caused for example by feature additions that are part of normal development efforts.

Finally, as this process step describes identification of technical debt propagation channels from historical data, it is evident that technical debt can exist without related software changes. For example, if an entity is created with principal for a new technical debt instance. No changes yet record alterations for this debt. Hence, arguably, identification of technical debt propagation channels requires historical—technical debt inclined change—data as it is the only system capable of recording how the debt has been realized.

3) Abstracting Channels to Models: Having identified the technical debt channels, modelling technical debt propagation corresponds to identifying a class T of technical debt channels $t \in T$ which demonstrate identical technical debt propagation capabilities $\mathbf{P}_T$, and abstracting this class to form a model M.

Technical debt channel t has a source $source_t = type(e_s)$, a destination $dest_t = type(e_d)$, and an information type $info_t = type((e_s, s_i) \rightarrow (e_d, s_j))$ which capture its propagation capabilities $\mathbf{P}_t = (source_t,\ dest_t,\ info_t)$. Hence, $t \in T \iff \mathbf{P}_t = \mathbf{P}_T$. For the software entities, the type was their context dependent definition (either reference-making-capable or referable; see Section III-A2), while the content forms the type for the information (see Section III-A1). Observation level fixing has ensured that the types observed for the entities and the information adhere to these requirements.

Using the Java Language Specification[1] as an example context, we may form a class of technical debt channels as follows: the $source_T$ is a "Method Signature" which is defined to be a referable type in this context. The $dest_T$, in this case, could be the "Method Invocation Expression" as it is capable of referring the source entity type. Regarding $info_T$, for explicit channels, the semantics of the context describe the types of systems that may deliver change between the previously typed source and destination entities (e.g. the Method Invocation Expression refers to the Method Signature either through direct dependency or through reflection). For implicit channels, a similar identification process is lacking, but fixing the source and destination entity types should reduce the number of information types to inspect. We demonstrate these processes in the next section.

Abstracting the model corresponds to removing all project-context information θ (e.g. names of specific Method Signatures) from the technical debt channels forming a class (i.e. $\forall t \in T$) to make the model applicable for all scenarios where the observed propagation capabilities $\mathbf{P}_T$ are identical. Hence, the abstraction of M corresponds to a reduction:

$$T \mapsto_\theta M \qquad (2)$$

B. Applying the Process

We provide initial validation for the technical debt modelling process described in Section IV-A by applying it to a technical debt instance captured in Figure 3. A bug from the Eclipse IDE (ID no. 73950 examined in [15]) covers multiple implementation entities, and it has been artificially expanded to highlight other possible areas of expansion from ongoing software development.

According to the previous process description in Section IV-A, the first phase of processing technical debt channels is observation level fixing (see IV-A1). This requires identifying the historical information and decomposing it so as to reveal the depicted software entities and changes between them. For

[1] https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

the Eclipse bug, we identify historical data and software entities via the bug report[2] and the corresponding fix commit[3]. The commit describes changes at the source code level, and this allows us to observe evolution sequences $e : (s_1, s_2, ..., s_n)$ at the level of single software entities.

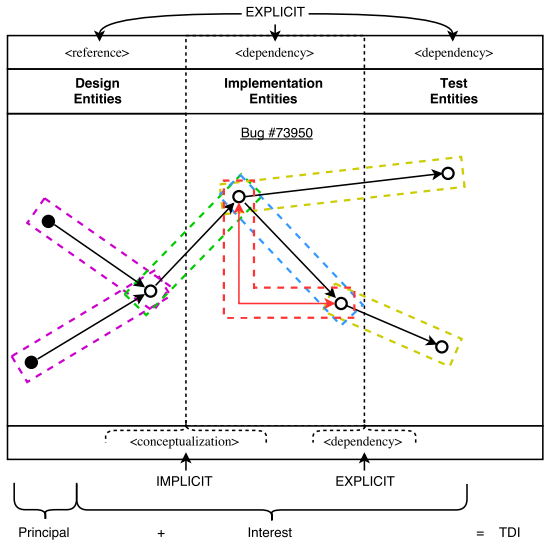

Fig. 3. A technical debt instance; the software entities associated to it, and the underlying technical debt propagation channels.

The second phase of the process identifies technical debt channels. Here, having ensured that software change information dissipates technical debt, the change information is decomposed and associated to producers and consumers. Figure 3 depicts a collection of possible associations for a single technical debt instance (a closer view, in the same color scheme, is provided in Figure 4). Backtracking the dissipation, the iterative process of finding producers and consumers should stop when software entities that produce the information about changes which overcome the root cause, the principal of the instance, are found. For the Eclipse bug 73950 in question, we associate the bug report's call for disposing `MemoryBlockAction` properly into changes in the fixing commit. Figure 4 is a transcript of the fixing commit, and it can be seen as a magnification into the implementation elements present in Figure 3. Lines in green and starting with a plus sign indicate addition, while the ones in red and starting with a minus sign indicate deletion. Numbered arrows indicate identified technical debt propagation channels—forming the propagation path for a technical debt instance—while the arrow colors indicate classes of channels with possibly similar propagation capabilities.

The Java context[1] (see Section IV-A3) applies for technical debt channel four (4). This is a pair $r = ((e_s, s_i), (e_d, s_j))$ where for source entity e_s `addMemoryBlockAction.dispose()` the type

[2]https://bugs.eclipse.org/bugs/show_bug.cgi?id=73950

[3]https://git.eclipse.org/c/platform/eclipse.platform.debug.git/commit?id=9d0372b5e5159743ef53b2ec0ddaf1bfbb58a0ce

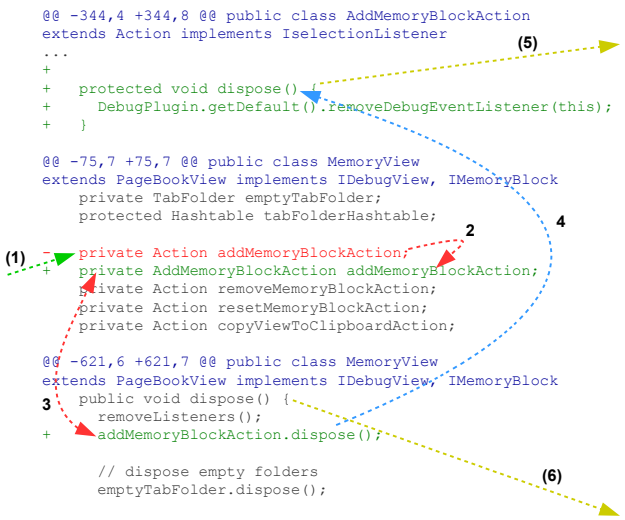

Fig. 4. Transcript from an Eclipse version commit, demonstrating implicit and explicit technical debt propagation channels with directions

$source_t = type(e_s)$ is a *method invocation*. The state s_i *statement creation* is likely the first one for e_s, and it has invoked another *statement creation* state s_j for the destination entity e_d `dispose()`, whose type $dest_t = type(e_d)$ is a *method declaration*. The information type $info_t = type((e_s, s_i) \rightarrow (e_d, s_j))$ is *invocation of a non-existent method declaration* as the method is created in the commit. Noteworthy is also that channels from one (1) to three (3) are implicit channels in Figure 4, and manual analysis is required to indicate these relationships [31]. In particular, the commit transcript alone cannot be solely used to decide e_s and e_d for channel three (3).

The final, third phase, of the process identifies a class of channels to abstract into a technical debt model. As discussed, this process relies on identifying similar types for the source and destination entities, and the information (see Section IV-A3). If we consider the channel four (4) to be the sole representative for its class, the abstraction results to a technical debt propagation model displayed in Table I (where the context removal θ disregards naming for e_s and e_d).

TABLE I
A TECHNICAL DEBT MODEL

Part	Definition
Source entity	Method Invocation[1]
Destination entity	MethodDeclaration[1]
Information	Invocation of a non-existent method declaration

Lastly, we may review the information properties described in Section III-B for the captured technical debt model (see Table I). The common property for technical debt channel information (i.e. either principal or interest) required that the software change indicates additional resource consumption. The *information* of the model adheres to this as the implementation of a method declaration is indicated. The unique

property of the information described if it accumulated either principal or interest for a technical debt instance. This required identifying if the additional resource consumption occurred in an entity that hosted the instance's root cause. The model's instances, the unique technical debt propagation channels, must be consulted for this. In the case of channel four (see (4) in Figure 4), an argument for interest accumulation can be made, if we interpret channels one through three with their entities to precede it in the instance's propagation.

V. Discussion

Below, we evaluate the strengths and implications of our solution in SectionV-A, and discuss potential challenges that may result from the implementation of our approach in Section V-B. We look to draw comparisons from our propositions, and the works of others introduced in Sections II and III.

A. Strengths and Implications

The most important strength of the proposed approach is the aspect that features the accumulated library of technical debt propagation channel classes. Through this mechanism models can be easily applied to estimate the technical debt propagation capabilities of new projects (i.e. we may assess models like the one in Table I for newly encountered similar components). This allows the project to: (1) expose possible propagation paths for newly developed entities by relating them to known source types, (2) provide enhanced explanation for problem targets by relating the target entities to known destination types, and (3) expose gaps in project communication by way of demonstrating the possible ways of propagation between project entities as the known information types. These strengths directly contribute to ongoing efforts of the research community mapping the technical debt cause, effect, and management landscape [32], [34], [35], and could have potential implication for practice.

The models also expose an interface that allows programmatic evaluation of the representations which is especially important from the perspective of automating information maintenance for constantly evolving projects. As models derived from the explicit channels capture technical debt propagation in contexts where the semantics are known, their evaluation can be implemented by means of static program analysis. For implicit channels, while the semantics can be unattainable and thus posing a challenge to full automation, the proposed approach collects the possible source and destination types which should allow for programmatic identification of their instances. This has the potential to produce a set of possible entry and exit points for technical debt propagation. The possibility of automating technical debt information maintenance has be recognized in previous works [9], [23], [25], [36], [37]. Such automation would arguably increase the effectiveness of technical debt management frameworks [7], [13]]. Furthermore, automating technical debt information maintenance would pave way for applying more established evaluation methods [8], [11] to manage accurately tracked technical debt instances.

In fact, while the approach proposed in this work is described somewhat disconnected from valuation efforts, there is no foreseeable obstacle to associating the models with value production (e.g. return-on-investment for expedited reparation of instances of the model in Table I). However, in order to associate the model with a cost value, the historical data needs to include value information (i.e. effort to overcome debt, such as hours-spent in refactoring an entity) and it must be decomposable together with the software change information (as described in Section IV-A).

B. Potential Challenges

Firstly, determining directions for, especially implicit, technical debt propagation channels can be difficult. In Figure 4, direction of the implicit channel depends on if the change is initiated by the type modification or invocation of the `dispose()` method. As technical debt channels are directed by definition, it is possible to model these cases from both directions. We anticipate that this could be a potential issue.

Second, the identification of classes as channels is based on type libraries. Given that the amount of type defining contexts (even when measured merely as the number of technologies available for implementation, design, and so forth) is remarkable, the amount of possible channel classes, combining source, destination and information type definitions, can be potentially numerous. To overcome this, arguably, a hierarchical channel taxonomy is required where the grouping dimensions exploit the pre-existing taxonomies available.

Third, two challenges relate to analyzing historical data in order to produce technical debt channels. Firstly, channel identification relies on distinguishing technical debt inclined change from the decomposed information. While the formal description of technical debt provides a basis for this, practical identification can be seen to rely on relating items to previously described instances of technical debt which is not exhaustive. Suovuo et al. have previously proposed examination of common change inducers as a partial solution to overcome this [31], and so this approach may reduce this burden. Secondly, historical data can only be used to reconstruct channels from where there is information regarding dissipation of technical debt. Hence, there can be channels that accumulate debt, but for which no data exist or the debt is never acted upon. The latter, arguably, captures debt that is almost invisible to the software project, but the former should be captured. Tracking of software projects' efficiency and addition of suitable documentation procedures to capture the missing evolution characteristics are fruitful avenues for research to overcome this issue.

Finally, Section II provided a review of existing research on technical debt. Whilst two approaches, one formal [22] and one hypothetical [21], address technical debt propagation, we note that neither of these could be seen to capture the various forms and ways of technical debt propagation. Rather, the approaches focus largely on the propagation's general characteristics and capabilities. We find this lack of differing approaches to technical debt modelling to be a challenge,

as it affects us properly benchmarking solutions in aiding meaningful comparisons.

VI. Conclusion and Future Work

This paper provided a theoretical description for technical debt channels as information mediums with producers and consumers. We also presented an approach for capturing technical debt channels, and identifying classes of channels, in order to abstract them into technical debt propagation models. In addition to advancing the technical debt research by providing theoretical basis for technical debt accumulation, the proposed method should be capable of delivering programmatically assessible models that allow automatic maintenance to be applied for manually identified technical debt information.

A most notable avenue for extending this work includes exploring mechanisms for identifying taxonomies and ontologies that describe types of technical debt information producers and consumers. Such mechanisms would facilitate the production of an accurate technical debt channel classification scheme. A direct application for such a classification scheme is the identification of previously overlooked technical debt management areas, and indication of enhancements for existing management solutions.

References

[1] C. Larman and V. R. Basili, "Iterative and incremental development: A brief history," *Computer*, vol. 36, no. 6, pp. 47–56, 2003.

[2] K. Schwaber and M. Beedle, *Agile software development with Scrum*. Prentice Hall PTR Upper Saddle Riverˆ eNJ NJ, 2002, vol. 18.

[3] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya *et al.*, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. ACM, 2010, pp. 47–52.

[4] N. Ramasubbu and C. Kemerer, "Towards a model for optimizing technical debt in software products," in *Managing Technical Debt, 2013 4th International Workshop on*. IEEE, 2013, pp. 51–54.

[5] P. S. M. dos Santos, A. Varella, C. R. Dantas, and D. B. Borges, *Visualizing and managing technical debt in agile development: An experience report*. Springer, 2013.

[6] J. Ho and G. Ruhe, "When-to-release decisions in consideration of technical debt," in *Managing Technical Debt, 2014 Sixth International Workshop on*. IEEE, 2014, pp. 31–34.

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

[8] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, and A. Vetrò, "Using technical debt data in decision making: Potential decision approaches," in *3rd International Workshop on Managing Technical Debt*. IEEE, 2012, pp. 45–48.

[9] F. Shull, D. Falessi, C. Seaman, M. Diep, and L. Layman, "Technical debt: showing the way for better transfer of empirical results," in *Perspectives on the Future of Software Engineering*. Springer, 2013, pp. 179–190.

[10] V. Singh, W. Snipes, and N. A. Kraft, "A framework for estimating interest on technical debt by monitoring developer activity related to code comprehension," in *Managing Technical Debt, 2014 Sixth International Workshop on*. IEEE, 2014, pp. 27–30.

[11] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 31–34.

[12] 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.

[13] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Da Silva, A. Santos, and C. Siebra, "Tracking technical debt - an exploratory case study," in *27th IEEE International Conference on Software Maintenance*. IEEE, 2011, pp. 528–531.

[14] J. Holvitie, V. Leppänen, and S. Hyrynsalmi, "Technical debt and the effect of agile software development practices on it-an industry practitioner survey," in *Sixth International Workshop on Managing Technical Debt*. IEEE, 2014, pp. 35–42.

[15] J. Holvitie and V. Leppänen, "Examining technical debt accumulation in software implementations," *International Journal of Software Engineering and Its Applications*, vol. 9, no. 6, pp. 109–124, 2015.

[16] N. Zazworka, C. Seaman, and F. Shull, "Prioritizing design debt investment opportunities," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 39–42.

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

[18] Y. Shmerlin, D. Kliger, and H. Makabee, "Reducing technical debt: using persuasive technology for encouraging software developers to document code," in *Advanced Information Systems Engineering Workshops*. Springer, 2014, pp. 207–212.

[19] T. Theodoropoulos, M. Hofberg, and D. Kern, "Technical debt from the stakeholder perspective," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 43–46.

[20] J. Holvitie and V. Leppänen, "DebtFlag: Technical Debt Management with a Development Environment Integrated Tool," in *Managing Technical Debt, 2013 4th International Workshop on*. IEEE, 2013.

[21] J. McGregor, J. Monteith, and J. Zhang, "Technical debt aggregation in ecosystems," in *3rd International Workshop on Managing Technical Debt*. IEEE, 2012, pp. 27–30.

[22] K. Schmid, "A formal approach to technical debt decision making," in *Proceedings of the 9th International ACM SIGSOFT Conference on Quality of Software Architectures*. ACM, 2013, pp. 153–162.

[23] N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2013, pp. 42–47.

[24] R. J. Eisenberg, "A threshold based approach to technical debt," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 2, pp. 1–6, 2012.

[25] D. Falessi, M. Shaw, F. Shull, K. Mullen, M. S. Keymind *et al.*, "Practical considerations, challenges, and requirements of tool-support for managing technical debt," in *Managing Technical Debt, 2013 4th International Workshop on*. IEEE, 2013, pp. 16–19.

[26] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" *Software Engineering, IEEE Transactions on*, vol. 34, no. 2, pp. 181–196, 2008.

[27] R. Schuppenies and S. Steinhauer, "Software process engineering metamodel," *OMG group, November*, 2002.

[28] A. Rochd, M. Zrikem, A. Ayadi, T. Millan, C. Percebois, and C. Baron, "Synchspem: A synchronization metamodel between activities and products within a spem-based software development process," in *Computer Applications and Industrial Electronics, 2011 IEEE International Conference on*. IEEE, 2011, pp. 471–476.

[29] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 2, pp. 77–131, 2007.

[30] *Merriam-Webster.com*. Merriam-Webster, 2016.

[31] T. Suovuo, J. Holvitie, J. Smed, and V. Leppänen, "Mining knowledge on technical debt propagation," in *14th Symposium on Programming Languages and Software Tools*. CEUR-WP, 2015.

[32] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice." *IEEE Software*, vol. 29, no. 6, 2012.

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

[34] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman, and F. Shull, "Organizing the technical debt landscape," in *3rd International Workshop on Managing Technical Debt*. IEEE, 2012, pp. 23–26.

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

[36] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali, "Searching for build debt: Experiences managing technical debt at google," in *Proceedings of the 3rd International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 1–6.

[37] J. Letouzey, "The SQALE method for evaluating technical debt," in *Managing Technical Debt, 2012 3rd International Workshop on*. IEEE, 2012, pp. 31–36.