Johannes Holvitie

Technical Debt in Software Development

Examining Premises and Overcoming Implementation for Efficient Management

Turku Centre *for* Computer Science

TUCS Dissertations
No 221, April 2017

Technical Debt in Software Development

Examining Premises and Overcoming Implementation for Efficient Management

Johannes Holvitie

To be presented, with the permission of the Faculty of Mathematics and Natural Sciences of the University of Turku, for public criticism in Auditorium Lambda on April 7, 2017, at 12 noon.

University of Turku
Department of Future Technologies
Vesilinnantie 5, 20500 Turku, Finland

2017

Supervisors

Prof. Ville Leppänen, Ph.D.
Department of Future Technologies
University of Turku
Finland

Adj. Prof. Mikko-Jussi Laakso, Ph.D.
Department of Future Technologies
University of Turku
Finland

Asst. Prof. Sami Hyrynsalmi, D.Sc. (Tech.)
Pori Department
Tampere University of Technology
Finland

Prof. Tapio Salakoski, Ph.D.
Department of Future Technologies
University of Turku
Finland

Reviewers

Prof. Kari Smolander, Ph.D.
Department of Computer Science
Aalto University
Finland

Asst. Prof. Alexander Chatzigeorgiou, Ph.D.
Department of Applied Informatics
University of Macedonia
Greece

Opponent

Prof. Philippe Kruchten, Ph.D.
Department of Electrical and Computer Engineering
University of British Columbia
Canada

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

ISBN 978-952-12-3524-5
ISSN 1239-1883

To my father

Abstract

Software development is a unique field of engineering: all software constructs retain their modifiability—arguably, at least—until client release, no single project stakeholder has exhaustive knowledge about the project, and even this portion of the knowledge is generally acquired only at project completion. These characteristics imply that the field of software development is subject to design decisions that are known to be sub-optimal—either deliberately emphasizing interests of particular stakeholders or indeliberately harming the project due to lack of exhaustive knowledge. Technical debt is a concept that accounts for these decisions and their effects. The concept's intention is to capture, track, and manage the decisions and their products: the affected software constructs.

Reviewing the previous, it is vital for software development projects to acknowledge technical debt both as an enabler and as a hindrance. This thesis looks into facilitating efficient technical debt management for varying software development projects. In the thesis, examination of technical debt's role in software development produces the premises on to which a management implementation approach is introduced.

The thesis begins with a revision of motivations. Basing on prior research in the fields of technical debt management and software engineering in general, the five motivations establish the premises for technical debt in software development. These include notions of subjectivity in technical debt estimation, update frequency demands posed on technical debt information, and technical debt's polymorphism. Three research questions are derived from the motivations. They ask for tooling support for technical debt management, capturing and modelling technical debt propagation, and characterizing software development environments and their technical debt instances. The questions imply consecutive completion as the first pursued tool would benefit from—possibly automatically assessable—propagation models, and finally the tool's introduction to software development organizations could be assisted by tailoring it based on the software development environment and the technical debt instance characterizations.

The thesis has seven included publications. In introducing them, the thesis maps their backgrounds to the motivations and their outcomes to the research questions. Amongst the outcomes are the DebtFlag tool for technical debt management, the procedures for retrospectively capturing technical debt from software repositories, a procedure for technical debt propagation model creation from these retrospectives, and a multi-national survey characterizing software development environments and their technical debt instances.

The thesis concludes that the tooling support, the technical debt propagation modelling, and the software environment and technical debt instance characterization describe an implementation approach to further efficient technical debt management. Simultaneously, future work is implied as all previously described efforts need to be continued and extended. Challenges also remain in the introduced approach. An example of this is the combinatorial explosion of technology-development-context-combinations that technical debt propagation modelling needs to consider. All combinations have to be managed if exhaustive modelling is desired. There is, however, a great deal of motivation to pursue these efforts when one re-notes that technical debt is a permanent component of software development that, when correctly managed, is a development efficiency mechanism comparable to a financial loan investment.

Tiivistelmä

Ohjelmistokehitys on uniikki tekniikan ala: kaikki ohjelmistorakenteet säilyttävät muokattavuutensa—otaksuttavasti ainakin—asiakasjulkaisuun asti. Yhdenkään projektiosakkaan tietämys ei kata koko projektia ja merkittävä osa tästäkin tiedosta karttuu vasta projektin suorittamisen aikana. Nämä ominaisuudet antavat ymmärtää, että ohjelmistokehitysala on sellaisten suunnitelupäätösten kohde, joiden tiedetään olevan epätäydellisiä—joko tarkoituksella tiettyjen projektiosakkaiden intressejä painottavia tai tahattomasti projektia vahingoittavia puutteelliseen tietoon perustuvia. Tekninen velka on konsepti, joka huomioi nämä päätökset sekä niiden vaikutukset. Konseptin tarkoitus on havaita, seurata ja hallita näitä päätöksiä sekä tuloksena syntyviä teknisen velan vaikutuksen alla olevia ohjelmistorakenteita.

Edellisen kuvauksen valossa ohjelmistokehitysprojekteille on erityisen tärkeää huomioida tekninen velka sekä mahdollistajana että hidasteena. Tämän vuoksi kyseinen väitöskirja perehtyy tehokkaan teknisen velan hallinnan fasilitointiin moninaisille ohjelmistokehitysprojekteille. Väitöskirjassa tarkastellaan teknisen velan roolia osana ohjelmistokehitystä. Tarkastelu tuottaa joukon premissejä, joihin perustuen esitellään lähestymistapa teknisen velan hallinnan toteuttamiselle.

Viisi väitöskirjan alussa esitettyä motivaatiota kiinnittävät ne premissit, joille ratkaisu esitetään. Motivaatiot rakennetaan olemassa olevaan teknisen velan sekä ohjelmistotekniikan tutkimustietoon perustuen. Näihin lukeutuvat muun muassa subjektiivisuus teknisen velan estimoinnissa, teknisen velan informaatiolle nähdyt päivitystaajuusvaatimukset sekä teknisen velan polymorfismi. Havainnoista johdetaan kolme tutkimuskysymystä. Ne tavoittelevat työkalutukea teknisen velan hallinnalle, velan propagoitumisen havainnointia sekä mallinnusta kuin myös ohjelmistotuotantoympäristöjen ja niiden velka instanssien kuvaamista. Tutkimuskysymykset implikoivat peräkkäistä suoritusta: tavoiteltu työkalu hyötyy—mahdollisesti automaattisesti arvoitavista—teknisen velan propagaatiomalleista. Valmiin työkalun käyttöönottoa voidaan taas edistää jos kuvaukset kehitysympäristöistä sekä niiden velkainstansseista ovat käytettävissä työkalun räätälöintiin.

Väitöskirjaaan sisältyy seitsemän julkaisua. Väitöskirja esittelee ne kiinnittämällä julkaisujen taustatyön aikaisemmin mainittuihin motivaatioihin sekä niiden tulokset edellisiin tutkimuskysymyksiin. Tuloksista huomioidaan esimerkiksi DebtFlag-työkalu teknisen velan hallintaan, retrospektiivinen prosessi teknisen velan kartoittamiselle versionhallintajärjestelmistä, prosessi teknisen velan mallien rakentamiselle näistä kartoituksista ja monikansallinen kyselytutkimus ohjelmistokehitysympäristöjen sekä näiden teknisen velan instanssien luonnehtimiseksi.

Väitöskirjan yhteenvetona huomioidaan, että teknisen velan hallinnan työkalutuki, teknisen velan propagaatiomallinnus ja ohjelmistokehitysympäristöjen sekä niiden teknisen velan instanssien luonnehdinta muodostavat toteutustavan, jolla teknisen velan tehokasta hallintaa voidaan kehittää. Samalla implikoidaan jatkotoimia, sillä kaikkia edellä kuvattuja työn osia tulee jatkaa ja laajentaa. Toteutustavalle nähdään myös haasteita. Eräs näistä on kombinatorinen räjähdys teknologia- ja kehityskontekstikombinaatioille. Kaikki kombinaatiot tulee huomioida mikäli teknisen velan propagaatiomallinnuksesta halutaan kattavaa. Motivaatio väitöskirjassa esitetyn työn jatkamiselle on huomattavaa ja sitä kasvattaa entuudestaan edellä tehty huomio siitä, että tekninen velka on pysyvä komponentti ohjelmistokehityksessä, joka oikein hallittuna on kehitystehokkuutta edistävänä komponenttina verrattavissa finanssialan lainainvestointiin.

Acknowledgements

I consider myself extremely fortunate having had the opportunity to pursue a doctoral degree through this unique path. I have had the opportunity to independently discover and pursue topics that captivate me, experience events and visit places I would have never been able to otherwise, and learn things that have vastly enriched my look on research and life in general. It is, however, due to none of these things that I consider myself so fortunate. It is the people around me, old friends and new, who have made this path a truly worthwhile one, and in the following I will do my best to acknowledge this.

My thesis is supervised by Ville Leppänen, Mikko-Jussi Laakso, Sami Hyrynsalmi, and Tapio Salakoski. In addition to being my supervisor, Ville also leads the Software Development Laboratory in which I have committed the research for this thesis. I wish to express my sincerest gratitude to Ville for the level of independence he has allowed for me while firmly steering my research to ensure its successful completion. I would like to thank Mikko-Jussi for his supervision, but also for encouraging and supporting me in applying to Turku Centre for Computer Science (TUCS) which led me to start on this path. Sami, above all things, thank you for being a great friend. It is not only that you have helped me on countless accounts and taught me most things I know about research pragmatics, but its the many discussions and events, especially outside work, you have involved me in that I will remember. To Tapio, an extended thank you for making so many things possible. Especially for sharing Ville's trust, and introducing me to challenging projects from which I have learned a great deal from.

Regarding the finalization of this thesis, I would like to thank Kari Smolander and Alexander Chatzigeorgiou for acting as pre-examiners. Their in-depth and constructive feedback allowed me to increase the quality of this thesis tremendously, and to do so in a very short period of time. I would also like to extend my thank you to Philippe Kruchten who has agreed to act as my opponent.

There are many people with whom I have had the pleasure of committing research for this thesis. Especially, I would like to mention and express my

gratitude to Sherlock Licorish (thank you for the tremendous and unyielding effort you have shown for our research, and for hosting me in New Zealand), Tomi 'bgt' Suovuo (thank you for all your help, the great talks, and for being patient with my ever-changing schedules), Antonio Martini (a friend you always meet on the other side of the planet), Jouni Smed, Stephen MacDonell, Jim Buchan, Rodrigo Spínola, Thiago Mendes, Olli Nevalainen, and Jurka Rahikkala.

Most of my daily work, I have done in the Software Development Laboratory. It is to my lab colleagues Jari-Matti Mäkelä, Timo Vasankari, Antero Järvi, Shohreh Hosseinzadeh, Sampsa Rauti, Jukka Ruohonen, Anne-Maarit Majanoja, Samuel Laurén, Eero Stürmer, Robert Siipola, and Lauri Koivunen that I owe thanks for the countless discussions, work related or not, that we have had on the way.

My path originally begun in the learning analytics team, and I would like to extend a special thank you to Erkki Kaila and Teemu Rajala who both were a tremendous help and provided excellent guidance to me. I would also like to thank Rolf Lindén (never could have figured that formula without your help), Einari Kurvinen, and Peter Larsson as well as everyone else working in the ViLLE-team, back then as well as now, who make the atmosphere there so welcoming!

I have also had the pleasure of meeting and befriending people through academic activities that are more or less extra-curricular in nature. Thank you to Sonja Hyrynsalmi, Kalle Rindell, Kai Kimppa, Tiina Nokkala, Antti Hakkala, Jari Björne, Janne Lahtiranta, and Olli Heimo for all the tremendous events, splendid conversations, and extraordinary distractions. A very special thank you also to all my friends (KS) who have continued to cheer me on throughout all these years. I must have forgotten to mention many, but know that this does not delude my gratitude!

To everyone at the Department of Future Technologies, a warm thank you for making it a great place to work in. Especially, to Sami Nuuttila, Pia Lehmussaari, and Heli Vilhonen for the daily help in keeping things running. My appreciation goes out also to the University of Turku Graduate School (UTUGS) and its coordinators. A special thank you to Elise Pinta, for all your commitment in our joint project and for the counselling you have provided.

In addition to the support from my home department, I would also like to acknowledge the significant financial support given to me by the Turku Centre for Computer Science (TUCS) as well as the UTUGS Doctoral Programme in Mathematics and Computer Science (MATTI). I must, unfortunately, continue to name organizations instead of people as I want to express my sincerest gratitude towards the Nokia Foundation, the Finnish Science Foundation for Economics and Technology (KAUTE), the Finnish Foundation for Technology Promotion (TES), and the Ulla Tuominen Foundation

who have supported the thesis. This support has had a drastically positive effect on my motivation and perseverance to carry out my work.

To end, I want to acknowledge my father, Jussi Holvitie. In all aspects of life, including my studies up to and including my doctoral thesis research, I have always been able to rely on my father's unwavering support. This is the single most important reason why this thesis exists today.

Turku, March 2017
Johannes Holvitie

List of original publications

I Holvitie, J. and Leppänen, V. (2013). DebtFlag: Technical Debt Management with a Development Environment Integrated Tool. In *Proceedings of the 4th International Workshop on Managing Technical Debt (MTD)*, pages 20–27. IEEE

II Holvitie, J. (2014). Software Implementation Knowledge Management with Technical Debt and Network Analysis. In *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on*, pages 1–6. IEEE

III Holvitie, J. and Leppänen, V. (2015). Examining Technical Debt Accumulation in Software Implementations. *International Journal of Software Engineering and Its Applications*, 9(6):109–124

IV Holvitie, J. and Leppänen, V. (2014). Illustrating Software Modifiability–Capturing Cohesion and Coupling in a Force-Optimized Graph. In *Computer and Information Technology (CIT), 2014 IEEE International Conference on*, pages 226–233. IEEE

V Holvitie, J., Licorish, S. A., and Leppänen, V. (2016b). Modelling Propagation of Technical Debt. In *Software Engineering and Advanced Applications (SEAA), 2016 42th Euromicro Conference on*, pages 54–58. IEEE

VI Suovuo, T., Holvitie, J., Smed, J., and Leppänen, V. (2015). Mining Knowledge on Technical Debt Propagation. In *Programming Languages and Software Tools (SPLST), 2015 Symposium on*, pages 281–295. CEUR-WS

VII Holvitie, J., Licorish, S., Spinola, R., Hyrynsalmi, S., Buchan, J., Mendes, T., MacDonnell, S., and Leppänen, V. (2017). Technical Debt and Agile Software Development Practices and Processes: An Industry Practitioner Survey. *Journal article under review*

List of co-authored original publications not included in the thesis

- Holvitie, J., Licorish, S., Martini, A., and Leppänen, V. (2016a). Co-Existence of the 'Technical Debt' and 'Software Legacy' Concepts. In *Technical Debt Analytics, First International Workshop on*. CEUR-WS

- Tamburri, D. A., Curtis, B., Fraser, S. D., Goldman, A., Holvitie, J., da Silva, F. Q. B., and Snipes, W. (2016). Social Debt in Software Engineering: Towards a Crisper Definition. *Dagstuhl Reports*, 6(4)

- Licorish, S., Holvitie, J., Spinola, R., Hyrynsalmi, S., Buchan, J., Mendes, T., MacDonnell, S., and Leppänen, V. (2016). Adoption and Suitability of Software Development Methods and Practices. In *2016 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE

- Rahikkala, J., Leppänen, V., Ruohonen, J., and Holvitie, J. (2015). Top Management Support in Software Cost Estimation: A Study of Attitudes and Practice in Finland. *International Journal of Managing Projects in Business*, 8(3):513–532

- Pulkkinen, P., Holvitie, J., Nevalainen, O. S., and Leppänen, V. (2015). Reusability Based Program Clone Detection: Case Study on Large Scale Healthcare Software System. In *Proceedings of the 16th International Conference on Computer Systems and Technologies*, pages 90–97. ACM

- Holvitie, J., Leppänen, V., and Hyrynsalmi, S. (2014). Technical Debt and the Effect of Agile Software Development Practices on It–An Industry Practitioner Survey. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 35–42. IEEE

Contents

Chapter 1

Introduction

'Technical debt' is a concept that describes the difference between the current and the optimal state of a software project and the effects caused by the difference. There are several causes for technical debt's emergence: lack of knowledge about applicable best practices or the future are exemplar scenarios that cause the project to deviate further from its theoretical optimum. The latter example also communicates that technical debt is often impossible to avoid as exhaustive knowledge about the future is generally impossible to accumulate. Further, the path to what might be considered optimal in the future can take a route which ignores the current optimum, and, as such, trade-offs are required. The difference from the optimum at any state (i.e., the technical debt) can affect the project in two ways. There is a positive short-term effect from technical debt, if the difference is accumulated consciously, due to for example trading off the product's build quality for its hastened release. A clear positive effect can be seen in the early market access or customer feedback that follows a quick release. The negative effect is felt in the long-term. Poor workmanship (i.e., the deviation from optimal) in existing project's software assets can have a deteriorating effect on later assets as they are established on top of the sub-optimal assets.

As the positive and negative effect of technical debt respectively influence a software project's efficiency and sustainability, it is evident that technical debt should be managed. Previous research on the subject has noted the sub-categories of technical debt identification, tracking, and information maintenance which need to be combined with effective decision making in order to successfully manage technical debt (Seaman and Guo, 2011; Guo et al., 2011a; Li et al., 2015). The field of technical debt research is still in an emerging state as Li et al. (2015) note, and as such there are a number of obstacles which need to be overcome prior to finding solutions for the previous sub-categories; allowing explicit implementation of technical debt management. The thesis in question, sets out to examine these obstacles.

Herein, however, the examination is done from a perspective that intends to maximize the cost-effectiveness of technical debt management for the adopting software organizations whilst having minimal effect on their development environments. This means that the thesis looks for ways to carry out the adoption while not introducing drastic changes to the organizations' existing software production arsenal: the used processes, methods, environments, tools and so forth. Through this examination the thesis aims to establish the role of *Technical Debt in Software Development* while *Examining Premises and Overcoming Implementation for [its] Efficient Management.*

Three chapters follow the current one. Chapter 2 is dedicated to defining the concept of 'technical debt'. The definition is built by reviewing associated software engineering domains and constructs. The role of technical debt with its effects and mechanism is established for the previous through examination of existing research. Notably, the software development life-cycle is reviewed with associated software assets, technical debt's emergence is established for it, and the state-of-the-art in technical debt management is reflected upon this. This discussion is distilled into five motivations for future research (*M1–M5* in Section 2.5). The motivations are derived from applicable rules in the software domain and the revised technical debt research, and they are presented as conditional hypotheses, steering the research approach and further limiting the working context.

Chapter 3 defines the research undertaken by the thesis and it is presented as research questions, and publications. The three research questions (*RQ1–RQ3* in Section 3.1) acknowledge the motivations and build sub-tasks for meeting the goal of *Examining Premises and Overcoming Implementation for Efficient Management* of Technical Debt. Each research question is justified via the referred motivations. The RQs are designed to be independent work packages having additional value delivery over meeting the grand goal, and they are constructed based on the related motivations that capture research on the area. The last part of Chapter 3 describes the seven publications (*P1–P7* in Section 3.3) included in the thesis to meet the set research questions. Certain research questions require decomposition to subproblems and overcoming them prior to meeting the research question itself. Thus, multiple publications contribute towards a single research question. The publications document independent research projects with their own research background, design, and execution, and, as such extend beyond the scope of the respective research question.

Finally, Chapter 4 concludes the thesis by summarizing the contributions made and establishing future work. The contributions are divided into research question contributions and supplementary contributions. The research question contributions correspond to the work carried out in the included publications that meet particular research questions, and thus aid in meeting the goal of *Examining Premises and Overcoming Implementation*

for Efficient Management of *Technical Debt in Software Development*. Supplementary contributions are divided into theoretical and practical ones. The theoretical contributions capture those results which can be seen to further the research in the technical debt domain while the practical contributions are seen to provide value when applied in the software industry. The last part of Chapter 4 ends the thesis by establishing future work for it. Committed research is reviewed to identify possible enhancements from future iterations while new avenues of research are acknowledged for the current findings.

Chapter 2

Technical Debt

Software development is concerned with the creation and modification of virtual assets. Combination of these software assets produces a software product which requires hardware in order to function. Many engineering disciplines work on virtual assets. For example, erecting a building foresees a number of architecture and interior plans prior to anything concrete taking place on the building site. Software development, however, never leaves the virtual domain. The finished product is virtual, and it functions in the real world only through the hardware selected to realize it. This fundamental difference allows for a number of special characteristics for software development.

First, all self-created software assets (i.e., the assets created by the same organization that utilizes them) can be accessed. Unlike in the physical world, where another physical obstacle can block and thus disallow access to another asset, software assets exist in the virtual domain, and the rules of the domain are provided by the applicable semantics. The semantics can, and often do, dictate access limitations (e.g., most programming languages define visibility of assets), but as the assets are self-created, we know where they can be accessed and how their representation can be changed. The term self-created is used to highlight that for 3rd party created software assets this access scenario does not fully apply: for 3rd party software, the access route for example is more obfuscated as the assets are, for example, often distributed as binaries or referred to via an Application Programming Interfaces (API). In the most restricted case, we may still opt to not use these software assets or to switch to using other ones.

Direct access to software assets leads to two important characteristics: software reuse and re-modification. Software reuse is the primary method of software development (Krueger, 1992; Selby, 2005). New software solutions are built by relying onto previous, partial, software solutions. For example, the bubble-sort algorithm (c.f., Knuth (1998)) is not re-implemented

every time it is needed, but rather a mature, previously made solution is just referenced with new parameters. Reuse is of importance as it allows both efficient software development to take place, but it also ensures that a particular functionality is always located in the same place in the software product. If the previous bubble-sort algorithm needs adjustments, only the one implementation must be modified instead of all the individual, independent implementations.

Software re-modification allows any previously self-created software asset to be re-modified at any given time. We may not repair the underfloor heating prior to removing the tiling on top, but we can repair our bubble-sort algorithm without touching the user interface. This ability of always being able to repair software assets—which are still under development (i.e., the developer still has access to them)—is an important premise to software development: it is a valid and often a proficient strategy to defer producing perfect software assets from the start. Rather, many software development methods produce proof-of-concepts which can be revisited later to adjust the level of completeness according to the actual need and severity of the asset (Budde et al., 1992).

Second, the definition of done for a software asset is dependent on the observability of the asset's implementation state. In the name of simplicity, focus is drawn to two distinct observers: the client and the developer. A software product is developed by the developer for the client. The client's definition of done encompasses if the requirements set forth by the client are completed to satisfaction. Hence, the software assets forming the product must be implemented in such a manner that they provide an adequate response to the requirements when the selected hardware realizes them. This definition of done requires very little, even no, observation of the assets' implementation state. The software developer is interested in meeting the client's requirements (assuming a business model where continued transaction with the client is desired) while maximizing return-on-investment (ROI). Hence, as the developer uses the implementation as a medium to meet the client's requirements, aims to produce a minimum viable product to maximize short-term ROI, and accounts for the software assets' maintainability to maximize long-term ROI, the developer's need and ability to observe the implementation state is considered high.

The differing definitions of done are an important premise for software development, as they drive the process to complete a software product. As the client's definition is a subset of the developers, the developer has room to consider different strategies for producing the software assets and to maximize ROI. What this free room also means is that a significant portion from the full definition of done is without an external body that would demand it to be met. The developer poses the implementation knowledge required to observe the full definition of done, but the developer has no

motivation to pursue it beyond the chosen strategy. Reviewing the previous characteristics, the following notions can be made.

1. The differing definitions of done allow software assets to exist in states of partial completeness without interfering the software development process.

2. The ability to re-modify self-created software assets at any time, allows software assets to have value even when they are not fully complete.

3. Software reuse can revalue the assets multiple times during the evolution of the software. The revaluation frequency can be notable as new relations are constantly introduced between software assets during development.

Technical debt is a concept which borrows from the field of economics to describe how the current sub-optimality in software assets affects the surrounding software product, the hosting development project, and the developer's and client's organizations as time progresses. Technical debt management acknowledges that these sub-optimalities should be managed similarly to monetary assets: their existence is inevitable and they can provide a loan like leverage when needed. However, like a loan, inability to manage these assets leads to increased interest payments which will exhaust available resources over time.

In the following, Section 2.1 will define the term 'technical debt' while briefly describing its research history. The concept is then discussed as part of software development in Section 2.2 to exhaustively describe the settings wherein technical debt resides while Section 2.3 provides a description for its effects. Section 2.4 provides an overview of current technical debt management approaches, and afterwards introduces a framework designed for introducing new management solutions. Finally, Section 2.5 derives motivations for further research from the previously discussed areas to lay the foundations for the work carried out in this thesis. To support a structured discussion for the remainder of the dissertation, the present section is ended with a short glossary of integral, upcoming or already discussed, terms. Each glossary entry points to the page(s) where the term is discussed further.

Glossary of Terms

software asset is any object that can be delimited from the software development that carries value for one or many of the software stakeholders (i.e, developer, client, end-user etc.). The delimitation is usually done based on *software asset type* or *software asset scope*. p. 5

software asset type is based on the *software development life-cycle phase* (e.g., requirements elicitation, analysis, design, implementation, testing etc.) in which the delimited object has been created in. Asset can be referred to by its type: *requirements elicitation asset*, *design asset*, *implementation asset*, *testing asset* and so forth. p. 14, 29, 32

software asset scope is based on the abstraction level from which this asset is observed. The abstraction level dictates the granularity and complexity of viewed objects. A crude division to *system*, *component*, *module* and *element* levels can be made. Here, for example, observing an asset at the component level refers to the asset being composed from one or several components specified during the software development life-cycle. p. 15

software asset state is the representation of the asset at a particular time. The asset state changes if the representation is altered. The alteration can be caused by a directed modification need to this asset or it can be caused by *software asset relation*s. In case of an **optimal state**, the asset's representation perfectly adheres to all requirements posed on it (i.e., context or asset relation requirements). When there are unmet requirements, the asset has a **sub-optimal state**. There is often considerable amount of subjectivity in the state interpretation. The **"good enough" state** is one in which the representation is, often due to practicality, interpreted to meet enough requirements to be considered complete. **Accommodation** may also affect the state of a particular asset. In this, the state representation of an asset is affected by the states of other assets to which this asset has relations to. p. 11, 20

software asset relation is a **dependency** between two *software asset*s. The relation is generally due to the representation of one asset having a **reference** to the representation of the other asset. The dependency can be explicit or implicit. **Explicit dependency** means that there are explicit mechanisms in place through which the existence of this dependency can be observed (e.g., semantics of a programming

language which describe how parts of a program can reference one another). If no such mechanisms are in place, the relation is an **implicit dependency** (e.g., when a developer interprets designs during implementation, the implementation assets have an implicit reference to the design assets). p. 63

software asset reuse is a form of *software asset* use where an existing asset–created originally to meet particular requirements–is used to meet new requirements. p. 5

software development is a comprehensive term that includes both the process of creating software assets to meet a particular need (i.e., the software development life-cycle), but also the required support functions (i.e., operations that do not directly contribute assets but allow them to be efficiently and systematically developed. These can range from configuration control operations to human resources). p. 14

software development life-cycle describes a set of **phase**s (i.e., development events and actions), *software asset*s expected from the phases, and their interconnections that have the intent of taking a client need and delivering as well as maintaining a set of software assets that meet this need. p. 14

software development context is a particular set of rules and restrictions that can be associated with the creation of *software asset*s of a particular *type* and *scope*. An example of a development context that limits the creation of implementation assets is the Python language. This language defines *element*s, *module*s and their *relation*s. As the Python language is a particular technology, this development context is also referred to as a **technology context**. p. 29

software development environment is a particular collection of *software development context*s. p. 36

software development project is an instance of the *software development life-cycle* that is realized in a *software development environment* by a particular selection of individuals. p. 16

software development organization is a structure that is capable of executing *software development project*s. This structure is generally comprised from individuals and support infrastructure. The physical manifestation of the organization can be a company or a unit within one. p. 16

technical debt item is a **technical debt instance** which can be delimited to a particular decision—trading-off development driving aspects—affecting particular *software asset*s and their *state*s. **Polymorphism** is a feature of technical debt items, and it refers to their ability to follow the software asset *relation*s into new *software development context*s and/or *software development life-cycle phase*s. As the underlying assets change, the technical debt item also polymorphs and manifests as new effects to the *software development project*. p. 22

technical debt propagation is the movement of technical debt within *software asset*s. Technical debt propagation within a set of software assets can be acknowledged by observing for the entire set a particular *technical debt item* to be affecting all the asset *state*s. p. 20

technical debt channel is a mechanism capable of advancing *technical debt propagation*. The channel has a **source**, a **destination** and a **medium** that takes the *technical debt item* from the source to the destination. The source is a *software asset* to which *relation*s can be formed, and the destination is a software asset that can form relations. The medium is the mechanism delivering information emergent from the source that induces a *state* change in the destination. In technical debt propagation, a technical debt item uses several technical debt channels to move in software assets and affect their states. p. 59

technical debt propagation model is a generalization of a mechanism capable of advancing *technical debt propagation*. The models are produced via abstraction from *technical debt channels*—the channels are instances of the models. The abstraction removes all *software development environment* related details from the channels and compares the this way abstracted *source*, *destination* and *medium* together. If all three remain identical through all compared channels, the channels are representing the same propagation model. A propagation model can be used to derive the possible technical debt channels for an encountered software asset, or it can describe assets that are sensitive to encountered information. p. 60

2.1 Definition

In his technical report to OOPSLA'92, Ward Cunningham described development of the WyCash portfolio management system and simultaneously coined the term 'technical debt':

> "... 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. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise."

Reviewing the paragraph, we identify a number of mechanisms that define technical debt. *"Shipping first time code is like going into debt"*, notes that software assets can be considered to carry debt immediately after their implementation; due to assumptions made during development or the surrounding software assets continuing to evolve immediately (i.e., the assets are modified, possibly independent, from the referring assets; invalidating the original design and implementation choices), as later noted. *"A little debt speeds development so long as it is paid back promptly with a rewrite"*, notes that, like the differing definitions of done, technical debt affected assets are allowed to exist, but they will start to negatively affect their surroundings, and *"The danger occurs when the debt is not repaid."* This and the following sentences convey the main mechanism for technical debt: sub-optimal software assets will inevitably accumulate 'interest' for as long as they exist.

A sub-optimal software asset can be considered to be a software asset that currently holds a state that differs from its optimal state. The optimal state of a software asset is, arguably, a state that can be assessed to be optimal from the point-of-view of all contexts that are relevant to this software asset. As most contexts have contradicting goals, the optimal state often remains a theoretical one. We discuss this matter further in Section 2.3.

With his report, Cunningham, arguably, wanted to highlight that the software reuse revalues existing software assets numerous times during continued development, and, hence, quality and maintainability of these assets can not be neglected. As Cunningham wanted to deliver this technical message to the management, he borrowed the 'debt' term from the financial world. At this point, however, the term had a mainly negative tone to it, and the lack of proper definitions for vital components disallowed further technical debt management research.

Technical debt has been a problematic concept to define, as Brown et al. (2010) note, and it is actually easier to describe this multifaceted concept

via its sub-components. For this, Tom et al. (2013) provide a consensus definition in the form of a systematic mapping study. The sub-components of technical debt are *principal*, *interest amount*, and *interest probability*. The principal of technical debt describes, for this particular instance of technical debt, the estimated cost of its removal. Using the bubble-sort algorithm as a running example: let us say that the developer implementing the algorithm opts to use a particular data type. The chosen data type is more elaborate than the algorithm's optimal implementation would require. The more elaborate data type requires cumbersome calls to be used which affects the algorithms implementation; making it more complex. Additionally, the interface to use the algorithm is affected: it requires users to call the algorithm with the chosen, more elaborate data type. If the algorithm's runtime is not affected ($O(n^2)$ performance (Knuth, 1998)), it is not immediately visible and apparent that the algorithm's implementation is less than ideal. The too complex implementation and interface of the algorithm would now correspond to the principal of this specific bubble-sort algorithm's technical debt.

Interest amount, or interest for short, of technical debt corresponds to the extra effort that is created as the consequence of this technical debt existing (Brown et al., 2010). Continuing with the bubble-sort algorithm: the clients of the algorithm have to adapt in some manner due to its sub-optimal, principal laden, state. In this case, the client's need to convert their data type to the more elaborate one prior to being able to call the algorithm's interface. Additionally, later additions to the algorithm are more laborious as a result of its complex implementation state (note: effort to modify the algorithm's original implementation is counted as principal). In both of the aforementioned scenarios, additional effort is accumulated due to the debt's existence and hence correspond to accumulated interest.

Interest probability, or interest realization probability, is the component that sets this concept apart from its financial counter-part. In the financial world loans have a realization probability of one (1); arrangements will be made to accommodate the loans existence (e.g., repayment or write-off). For technical debt, however, this is not the case. The realization of further interest depends on the realization probability of the interest scenarios (Seaman et al., 2012). In case of the bubble-sort algorithm this would correspond to the probability that further clients use the algorithm or that the algorithm's implementation is extended. Existence of the interest realization probability is an important premise for technical debt management, as it allows for debt prioritization, and even planned debt neglecting. This matter is further probed in Section 2.4.

A very recent definition for 'technical debt' comes from the Dagstühl Seminar series, and it captures the previous sub-component definitions rather concisely. The thesis author had the opportunity to lead the discussion on

creating the *16162 Definition of Technical Debt* which is published as part of the seminar proceedings (Avgeriou et al., 2016). The 16162 marks the running number identifying *The Dagstuhl Seminar 16162 - Managing Technical Debt in Software Engineering*[1] which was held in Schloss Dagstühl in Germany from the 17$^{\text{th}}$ of April, 2016 onwards. As a result of the discussion held, to which a number of distinguished technical debt researchers from both the academia and the industry participated, the definition can be seen to represent the field's consensus rather exhaustively. It is stated as follows:

> "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."

Reviewing the definition, we note that technical debt is defined to be a *"design or implementation construct"*, a structure which we can locate from one of the software project's assets (discussed by the following Section 2.2). The construct is *"expedient in the short term"*, meaning that knowingly or unknowingly (discussed in Section 2.3) it furthers the development, *"but [it] sets up a technical context"*. Here, the context marks that we are not limited to a particular area (e.g., the construct's originating asset; discussed in Sections 2.3 & 2.5.2). Then, the technical context *"can make future changes more costly or impossible"* where the conditional marks that there is a realization probability, and the effect is limited to future changes, for which the definition describes negative effects ranging from lowered efficiency to incapability. Finally, the last sentence first marks the fuzziness and uncertainty that the hosting software project needs to work with in defining *"technical debt [as] an actual or contingent liability"*, and further continuous to scope its effects in stating that the *"impact is limited to internal system qualities, primarily maintainability and evolvability."* This is an important notion, as we can see that something which can be created in early design stages of a project may persist and effect stages much later into the project; possibly with completely disconnected staffing as the following section describes. Of note is also that the sentence communicates the immediate effects to system qualities. These may act as proxies and affect other areas. Even the client's system operation (Morgenthaler et al., 2012).

[1] `https://www.dagstuhl.de/en/program/calendar/semhp/?semnr=16162`

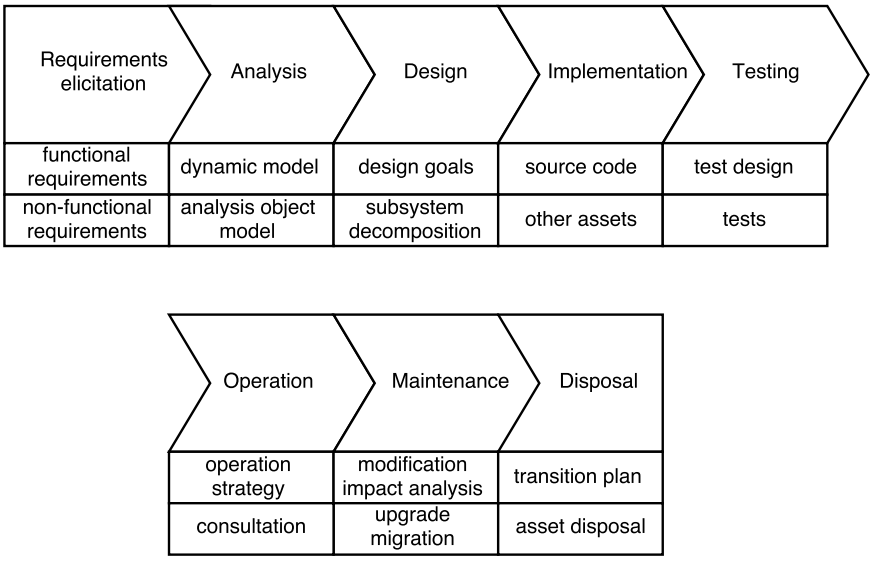


Figure 2.1: A generalization of a software life-cycle with the phase name on top and the produced software artifacts immediately below each phase. The upper section documents the phases which complete the product (Bruegge and Dutoit, 2004). The lower section documents the phases the completed product undergoes (ISO/IEC 12207:2008)

2.2 Software Development

Technical debt emerges, affects, and is controlled by software projects. To fully understand this setting, the following examines the basics of software development, encompassing the software life-cycle, the participating stakeholders, and the applicable development methods. Section 2.3 will then review the ways with which technical debt can emerge, propagate, and thus affect the project.

There are numerous interpretations of the software life-cycle available (Bruegge and Dutoit, 2004; IEEE 1074:2006; ISO/IEC 12207:2008). A generalized view is provided in Figure 2.1. The figure encompasses the eight main phases of software development. Phases that precede these are called pre-development phases while the phases succeeding these are called post-development phases. Software development starts with the *requirements elicitation* phase in which a problem (e.g., a client's request or a new business venture) is converted into a coherent and unambiguous set of requirements which are captured in the produced requirements elicitation software assets as functional and non-functional requirements. Functional requirements de-

scribe what should occur when the system-under-development interacts with its surroundings. For example, the bubble-sort algorithm should take a list of numbers and return it sorted. The non-functional requirements describe qualitative characteristics for the functional requirements. So, as an in-place sorter, the bubble-sort algorithm should not consume more than a constant amount of memory (Knuth, 1998).

The *analysis* phase takes the requirements and converts them into a static and dynamic description of the target system on an abstract level. The static description documents the objects, the parts of the system, that are required in fulfilling the requirements while the dynamic description documents how they interact to fulfill the requirements. The *design* phase realizes the abstract model. The phase prioritizes the non-functional requirements and their trade-offs form the design goals. These drive the selection of suitable realization techniques which then lead to subsystem decompositions of varying granularity (e.g., from high level architectural models to low level descriptions of singular classes and methods). The *implementation* phase is responsible for the construction of the designed system. Following the designs, the implementation phase develops source code and other assets (e.g., graphics for user interfaces) which, when ran on hardware, should produce an answer to the original problem. The, *testing* phase is responsible for both verifying and validating that the developed software, the system, produced an answered to the problem. Verification ensures that the requirements, the models, and the designs truly capture the problem. Validation ensures that the implementation assets developed from the previous software development life-cycles' assets truly produce an answer to the problem; *the observed behaviour corresponds to the expected behaviour* (ISO/IEC 12207:2008; Bruegge and Dutoit, 2004).

The *operation* phase takes place on a completed software product. The operation strategy can be seen to account for details which affect the product's operation. These include descriptions for correct operation of the product in its environment and the mechanisms that need to be in place for gathering problem and feedback reports. As the product operates in its environment, the developer is required to provide support for the users in addition to resolving possible problem situations. These events correspond to the consultation provided in this phase. The *maintenance* phase discovers modifications for the software product under operation. Impact analysis is carried out for the modifications in order to understand how the change could affect the organization, operations, or interfaces dependent on the product. The upgrade will then be implemented and migrated to the operated product. Relevant documents and other participating bodies are informed about the change according to the impact analysis. Finally, the *disposal* phase marks the end of the software product's life-cycle. A transition plan identifies how the environment in which the product operates will

Table 2.1: Software stakeholder groups and roles within them (Bruegge and Dutoit, 2004)

Management	**Development**	**Consultant**
Organization Management	Analyst	Client
Project Management	Architect	Technical Consult
Team Lead	Tester	End-User
	Designer	Domain Specialist

be affected, in both long- and short-term, when termination takes place. According to this, a set of disposal constraints are derived (e.g., certain assets may be deemed useful or they are dependent upon by other projects) which describe how the project's assets will be disposed.

There are a number of stakeholders that are involved in realizing a software project. A crude division into three categories is provided in Table 2.1. The *management* stakeholders are concerned with strategizing how the software project will be completed, and require active monitoring of the different phases in order to derive information on the current project state. New decisions are made based on this information to steer the project towards the chosen strategy. The *development* stakeholders are responsible for realizing the software development phases and thus their activities correspond to the previously explained phases. The *consultant* stakeholders come from outside the organization-developing-the-software and they provide expertise from areas that the organization can not cover. Most notably, as the software product is generally developed to meet a problem in a completely unrelated domain it is vital to consult the client and the end-user as they are the only roles with in-depth knowledge from their working environment. For example, the bubble-sort algorithm could be used as part of a software product that will handle medical records in a hospital environment.

A software project utilizes a software development method or a combination of methods to structure and carry out work specified in the software lifecycle. A software development method describes processes, practices, and/or roles for pursuing completion of the work. Two rough categories exist for the pursuation: plan-driven (or sequential) and iterative-incremental. Plan-driven software development methods were initially adapted from classical engineering domains (e.g., mechanical engineering as software was produced tightly coupled to particular mechanical systems (Royce, 1970; Larman and Basili, 2003)) where a detailed plan was pursued prior to implementing anything concrete. The benefit of these methods is that they can ensure, in the planning phase, that every requirement will be met. This should increase confidence in committing to an expensive project. It has commonly been noted that the software equivalent development method of this is the *Water-*

fall. It gets its name from going through the life-cycle phases in Figure 2.1 one-by-one, like water descending a waterfall. The problem is that software projects are highly complex and obfuscated which means that it is almost impossible to exhaustively know what is required prior to actually having working software assets—on to which it is easy to indicate what should be delivered next. But since the plan-driven methods do not account for the plans to change, it is extremely expensive to revisit the earlier phases. It has been later argued that the author of the Waterfall method did already suggest looping back to previous phases in the life-cycle, but due to strong association with the waterfall metaphor, the article by Royce (1970) was misinterpreted to mean a single consecutive execution of the phases (Larman and Basili, 2003; Bossavit, 2012).

The iterative-incremental methods organize work in a manner that expects changes to take place. These methods became popular after the previously mentioned problems with plan-driven methods became evident (Larman and Basili, 2003). For these methods, the commonality is that a single run of the software life-cycle (especially the phase on the upper row) in Figure 2.1 produces an *increment* for the software product. This is generally not a finished product, but it has to function; it has to meet the definition of done for those functional and non-functional requirements that were set out to be completed in this increment. This allows the testing to verify and validate a much smaller portion of the system, and the feedback received (e.g., from the client) is used to steer development in upcoming runs, or *iterations*, of the software life-cycle. Since the release of the Agile Manifesto by Beck et al. (2001), which collects many of the premises for iterative-incremental methods, these methods have generally been referred to as the Agile or the Lean methods.

In partitioning and scheduling the work into increments completed during iterations, Agile methods provide an excellent tool for complexity management. However, this very tool also enables technical debt to be accumulated as development organizations intentionally or unintentionally trade-off certain development driving aspects (e.g., internal quality) for others (e.g., core functionalities) in the increments. The following section examines this closer.

2.3 Emergence and Effects

As described in Section 2.1, technical debt consists of sub-optimalities in the software assets which affect the software project as the system-under-development evolves. We noted from Section 2.2 that software development is a multifaceted undertaking wherein numerous inter-connected software assets are produced to assist the work of several different stakeholder groups.

This section discusses the primary methods for technical debt to emerge and to affect the software development.

As technical debt consists of sub-optimalities in software assets, it emerges as an end result of the work done in their development. The work can accumulate technical debt either unintentionally or intentionally according to McConnell (2007). *Unintentional* technical debt is accumulated when the developing organization is not aware of the created sub-optimality in the asset. Examples of this for the bubble-sort algorithm would include poorly written code that causes confusion later on, unsuccessfully chosen design that makes testing harder, or use of a 3rd party bubble-sort that does not adhere to the software project's assumptions. As the debt is accumulated unintentionally, and it is the end result of non-strategic decisions, this type of debt is extremely dangerous to the software project. The debt remains hidden until it is identified, and, depending on the evolution speed, might have already accumulated a notable amount of interest at the time of discovery.

Intentional accumulation of technical debt is the end result of strategic decisions to produce sub-optimal assets. These decisions generally prioritize certain development driving aspects over others. Examples of this would be a decision to implement the bubble-sort algorithm without accompanying unit tests which prioritizes functional delivery (e.g., to meet a set deadline) over long-term maintainability of this asset. Intentional debt can be considered less dangerous as it is visible to the organization, and this type of debt is very close to its financial counterpart. The example trade-off (i.e., taking of the loan) can pursue delivering a functioning sub-product to the customer. This allows the customer to provide feedback on this particular functionality and hence allows steering future development. Additionally, a contract may call for sub-product delivery to secure additional funding, in which the aforedescribed loan might be vital to the project's continuation.

Fowler's (2009) "Technical Debt Quadrant" is adapted in Table 2.2, and it can be seen to add an extra dimension to explain the emergence of technical debt. The *inadvertent-deliberate*-axis is almost analogous to the previously described unintentional and intentional pair, while the *reckless-prudent*-axis describes how or in which circumstances the debt is accumulated. Recklessness implies a lowered state of strategization which can be seen leading to situations wherein decisions that accumulate technical debt are either deliberate or inadvertent, but in either case they do not consider all implications prior to making the decision. The end result of this is either that some technical debt remains undocumented (i.e., the inadvertent) or some can be documented (i.e., the deliberate) but its full ramifications remain unknown. For the prudent technical debt, the deliberate case is analogous with the example given for intentional debt before. The prudent-inadvertent case, however, is more interesting. Fowler refers

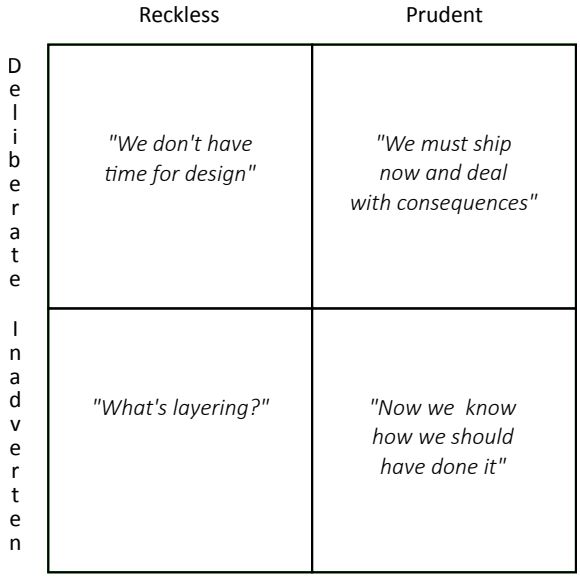


Figure 2.2: The technical debt quadrant adapted from Fowler (2009)

to Ward Cunningham's explanation[2] here, as this type of debt is usually identified through the course of learning. A software product is generally a very complex structure for which the design evolves continuously. Hence, it is usually only after the product is complete, when all hidden problems and changed requirements have been identified, that the software development can review the product and hence argue for what would have been the best design for this software product. Hence, the hindsight, the learning experience here, indicates the prudent-inadvertent debt.

Technical debt affects software development in the form of resource use inefficiency. As discussed earlier, software is built by relying onto earlier solutions. A more complex solution is accumulated by referring to previously created, less complex, solutions. This applies to both the implementation phase, wherein source code refers to earlier libraries and classes to produce functionality, but, also, to other phases in the software life-cycle as for example a sub-system design relies onto referred sub-systems providing the services their interfaces declare. The mechanism of resource use inefficiency for technical debt emerges from this reuse.

In reviewing these emergence and effect mechanisms for technical debt, one needs to also consider their closeness to the concept of 'software legacy'. Assets which can not be subjected under the same maintenance, and management procedures as newly created ones are considered to be software

[2]Ward Cunningham explains the 'debt' metaphor from the OOPSLA'92 technical report `https://www.youtube.com/watch?v=pqeJFYwnkjE`

legacy (Fowler et al., 1999; Feathers, 2004). In practice, this can mean that the software development organization's testing suite can not accommodate the assets, or the intellectual property needed for the assets' development is no longer available due to developer turnover. As such, 'software legacy' indicates asset sub-optimality and additional resource consumption which brings it very close to 'technical debt'. The difference, arguably, is in the origins. Software legacy is the end result of software degradation overtime. Mainly, due to the surrounding environment developing while the software remains static (Bennett, 1995; Dayani-Fard and al, 1999; Sommerville, 2008). It is a general state of software. Technical debt is the end result of singular actions; intentional or unintentional (McConnell, 2007; Fowler, 2009). This leads to technical debt being accumulated in increments with a size that is proportional to the action and the asset affected by the action. Hence, the methods for discovering, tracking, and maintaining 'software legacy' and 'technical debt' work from different premises even though the effects and mechanisms involved can be very similar. To further the interchange of information between these two domains, the author has co-started an investigation into the similarity of the two concepts in Holvitie et al. (2016a).

The state of each solution dictates its functionality. If a solution deviates from its optimal state there is an increased chance that the sub-optimality is reflected in its functionality (Schmid, 2013). As an asset's client (e.g., a controller method calling the bubble-sort algorithm to sort a list) has to adhere to the referred asset's interface in order to use the asset, the sub-optimality in the referred asset will affect the referring asset; given that the referred asset's interface reflects its sub-optimal state. This effect in the referring asset is the work done to accommodate the sub-optimal state in the referred asset. This means that extra effort went into the creation of the referring asset, its state is now also sub-optimal, and hence it consumed more resources than its implementation could have consumed. This phenomenon is called *technical debt propagation*, and it accumulates interest for the technical debt in software development projects. For the previous, it should be noted that the optimal state is generally impossible to achieve, and, rather, a "good enough" state is generally the one that is accepted in practice. The 'optimal state' concept is however required to reduce subjectivity and introduce formalism; especial in the research for new technical debt management procedures (Schmid, 2013). Differences in applying the previous states are discussed further as part of limitations in Section 4.2.

As we noted, technical debt spans the whole software life-cycle and is hence capable of propagating within the various software assets using the reuse mechanism. A further problem relies in the various stakeholder groups that are associated to the assets. As stakeholders generally represent expertise from non-overlapping domains, there is also an associated communica-

tion problem that comes with having very different linguistics and concepts. Hence, the technical debt that propagates from an asset to another asset that resides in a different stakeholder domain can be seen to have a higher danger of dropping out of scope. In these cases there is a danger that a problem presents itself in the client domain, but the developer is unable to promptly address the issue as the root cause domain can not be identified (Klinger et al., 2011).

As the resource use inefficiency increases, the software development faces a number of difficulties of increasing severity. Most evidently, less work can be done in consecutive iterations as more time has to be committed to dealing with technical debt when implementing new functionality. In Agile software development, this is not ideal, as the bloated resource estimates for singular tasks make the developing organization more rigid: smaller work-packages are easier to shuffle around by taking on more or dropping them if the situation changes. As Agile software development is all about the ability to react to changes, technical debt poses a considerable hindrance to the method's concept.

Further, as we have discussed that unintentional, and especially inadvertent-reckless, debt is difficult to track, due to the developing organization creating it unbeknownst to itself, there will be an increasing portion of the software development resources for which no estimates can be presented. This further decreases the organization's capability to estimate its capabilities. Also, the organization may think that it is able to take on more debt, as only some of its current debt is visible to it (Tom et al., 2013). Finally, as technical debt controls an increasing amount of software development resources, there comes a point in which more resources are consumed for taking care of technical debt rather than producing additional functionality. This point is called the technical debt *bankruptcy* (Tom et al., 2013), and it means that external resources will be required to bail the project (e.g., a refactoring strike-force or project scrapping and restart).

It should be noted that there are challenges related to using a metaphor, especially with this level of immediateness. Tom et al. (2013)'s results indicate that technical debt is not only a metaphor, but it also indicates a monetary cost. Further, discussions have taken place[3,4] in which the applicability and clarity of the metaphor have been questioned. Additionally, Booth (1978) notes that a good metaphor should be *accommodated to the audience*. As this claim is yet to be made for 'technical debt', which comes from the financial domain, especial care should be taken when using extensions of the metaphor like *bankruptcy* or even *interest*. Persons with differing backgrounds can interpreted these differently (Gildea and Glucksberg, 1983).

[3] `http://powersoftwo.agileinstitute.com/2009/03/ward-cunninghams-debt-metaphor-isnt.html`

[4] `https://www.infoq.com/news/2011/07/death_of_tech_debt`

Table 2.2: A Technical Debt Item (TDI) (Guo et al., 2011a)

Field name	**Value**
ID	Running index
Date	Date and/or hosting assets' revision
Type	Technical debt type
Location	(Hierarchical) location description
Description	Description for the item's emergence
Principal	Estimate for the item's principal (required effort e.g., in man-hours or cash)
Interest amount	Estimate for the item's interest
Interest prob.	Estimate for the item's realization

2.4 Management

It has become evident from the previous sections that management of technical debt is a vital component of software development. Especially, since technical debt is a global phenomenon which manifests in all software development life-cycle's regardless of their intrinsic characteristics. Due to its multifaceted nature, technical debt is difficult to manage. In the following, we will be overcoming a technical debt management framework which can be considered a basis on to which concrete management approaches can be introduced and which allows combination of these approaches into larger management suites that match the particular software life-cycle in question. Further, a review of existing, concrete, approaches is provided in order to establish the current state-of-the-art in technical debt management.

The technical debt management framework (TDMF) was introduced by Guo and Seaman (2011), and it encompasses three phases: identification, estimation, and decision making. Identification captures the software development's technical debt while estimation refines this information in order to enable decision making based on it. The following reviews these phases in more detail.

The technical debt identification phase is responsible for capturing the technical debt for the software development, and it does this by accumulating a Technical Debt List (TDL) which consists of Technical Debt Items (TDI). A TDI captures a particular technical debt which can be limited off from the system and handled by a single person or a team; for example, overtly complex interface for accessing the bubble-sort algorithm. Identification of a TDI corresponds to filling in the fields listed in Table 2.2. The *ID* field is a running index, very similar to an identifier given to software bugs (see e.g., Eclipse BugZilla[5]). This allows for easy referencing via

[5] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=465206`

(hyper)linking between TDIs. The *date* field describes when the TDI has been identified, or, preferably, the particular version for the software assets hosting the TDI. This information is of utmost importance as it ties the other fields of the TDI to a particular state in the hosting assets' evolution. Section 2.2 discussed the nature of software development wherein existing assets are created via reference to other assets, and this evolves the software project. Hence, it is important to note the particular state in the evolution, for which the captured TDI information is valid since the evolution will affect the validity of the captured information (Schmid, 2013).

The *type* field for a TDI (see Table 2.2) captures the generic category to which this technical debt item can be seen to belong. This categorization is based on the TDI's location and effort estimates. Various types of technical debt have been identified (Alves et al., 2014; Tom et al., 2013). In general, the type describes the software life-cycle phase or the assets produced by the phase which are affected by the debt in this TDI (e.g., architectural, testing, design, and social debt). The type is important to identify, as it describes which governance methods are applicable to the TDI. Grouping can also be considered for similarly typed TDIs for more efficient management. Noteworthy about the *location* field is that, in addition to describing the software assets affected by the TDI, the field also imposes a global interpretation rule on to the identification. If a particular asset is indicated as the host of the TDI, and the asset is part of a hierarchical structure which allows sub-assets to be identified for it, then, from a management viewpoint, we may decompose the TDI on to the sub-assets. This is important to understand when indicating the location because the sub-assets may call for different management procedures and they may evolve independent from one another as Marinescu (2012) notes.

The *description* field shortly describes the reason behind this TDI's existence. While less utilized in the management, it is important to note that this fields provides an opportunity for the TDI's author to justify why the debt was accumulated. Further, if the description is difficult to produce or it is missing, this can be indicative of the debt being accumulated inadvertently and/or recklessly. For some of these cases, it is debt that is discovered later, detached from reasoning. The estimates provided in the *principal*, *interest amount*, and *interest probability* (as man-hours, cash, or other effort valuation) have definitions that are analogous to those given in Section 2.1. However, they are limited to describing the effects, on resource consumption efficiency, that are caused by the technical debt residing in the announced location: the principal describes the effort to convert the hosting assets to an optimal state as far as this TDI is concerned. The interest amount describes the additional work that clients of the TDI's hosting assets will need to accumulate due to using the assets. The probability describes how likely it is that there will be new clients. In providing the estimates the

considered future time period is usually the iteration length which is usually also the update frequency for the Technical Debt List. Finally, the estimation accuracy is usually based on convenience (assumably this is the case as documenting the instances can not be laborious in order not to discourage recording them) and knowledge about the upcoming iteration. The authors of the TDMF suggest a three tier scale of low, medium, and high for both the effort and the probability (Seaman and Guo, 2011).

Reviewing the aforedescribed fields, we note that many of them require subjective input, and, hence, identification of technical debt is in most cases a manual task (Holvitie and Leppänen, 2013; Seaman et al., 2012; Marinescu, 2012; Guo et al., 2011a). However, for some limited technical debt instances, automatic identification can be used (e.g., general rule violation identification in SonarQube's Technical Debt[6] and SQALE (Letouzey and Ilkiewicz, 2012) plugins, or rule aggregation based identification of God Glasses in (Zazworka et al., 2011)). But even in these cases manual input is required to fully complete the instances fields. Especially, for the location, rule based identification may match several sub-assets of a single technical debt instance. Subjective input is required to compile them into efficiently manageable TDIs' locations.

From the previous, it is evident that technical debt identification has a lot in common with the domain of software quality assessment. The identification, in several cases can be—reduced or enhanced—to be a software quality assessment problem: design smell identification (Suryanarayana et al., 2014; Fowler and Beck, 1999), design flaw acknowledgement through metric values (Bansiya and Davis, 2002), design evaluation through review practices (Parnas and Weiss, 1985), or assessment of refactoring opportunities for software implementation assets (Tsantalis and Chatzigeorgiou, 2009) to name prominent examples. Arguably, however, the suitability of the quality assessment methods should be scrutinized, especially from the point-of-view of efficiency: even if an established quality assessment method produces accurate results, it may take a lot of additional effort to, for example, combine or decompose the matches into meaningful TDIs and to assess what the TDIs' interest probability amounts to be in this particular software development environment. It is for this reason that further research should be committed to look into bridging the two domains of technical debt identification and software quality assessment in a way that produces overall feasible solutions for software development organizations to adopt (we note a similar bridging need between software legacy and technical debt in Holvitie et al. (2016a)).

The technical debt tracking and estimation phase is responsible for maintaining the technical debt information for the software project by updating the TDL through the TDIs forming it. As each TDI is associated

[6] `http://www.sonarqube.org/evaluate-your-technical-debt-with-sonar/`

to particular states of particular software assets, it is the evolution of either that also affects the TDI. Exception to this is the estimate for the TDI's interest as this depends on the interconnectivity evolution of the hosting assets rather than the assets themselves. That is, new assets may be introduced during continued software development; evolving the interconnectivity of the hosting assets. The relations introduced by these new software assets increase the potential interest effort, if the new, interconnected assets can be affected by the technical debt in the hosting assets.

The tracking phase is more welcoming to automated solutions. This is due to the fact that the interconnectivity evolution of software assets, within the same domain, is dictated by the semantics of that domain. If we have identified a technical debt instance in our example bubble-sort algorithm, then we may automatically derive an estimate for the interest based on the asset relation semantics of the bubble-sort's implementation technology. Given that the semantics are available for automatic assessment and we associate each reference with a meaningful interest value (e.g., in man-hours or cash). While for all implementation asset technologies the semantics are available (as they are responsible for realizing the functionality of the implementation), the meaningful interest values are less trivial to attain. As these values are affected by subjective actions in particular software development environments, we would need to retrospectively analyse similar cases to produce meaningful average cases and even then decompose the averages to the level of single relations for association.

A further limitation can be observed for the semantics. We have discussed that assets between software life-cycle phases are associated to each other in Section 2.2. As there are no semantics available that describe referencing between assets of different domains, an automatic estimate based on the interconnectivity observed only in one domain can be considered rather optimistic. Tracking and estimation solutions that are capable of capturing single domains are still of utmost value, as many software organizations generally specialize in only a handful of different domains and, hence, their accumulated value is concentrated only to these few domains. Exemplar tools of this approach are the Blaze (Singh et al., 2014) which continuously evaluated code comprehension metrics in the Visual Studio integrated development environment (IDE), the aforementioned SonarQube plugins Technical Debt and SQALE, and the PMD[7] and FindBugs[8] static code analyser tools, in addition to the inFusion[9] tool which also does static rule evaluation. For the remaining TDIs, manual input is required to identify how the software assets associated to a particular TDI have evolved. The result of

[7] `https://pmd.github.io/` (Dixon-Peugh et al.)

[8] `http://findbugs.sourceforge.net/index.html` (Hovemeyer et al.)

[9] `https://www.intooitus.com/sites/default/files/AssessingDesignQuality.pdf` (Marinescu et al.)

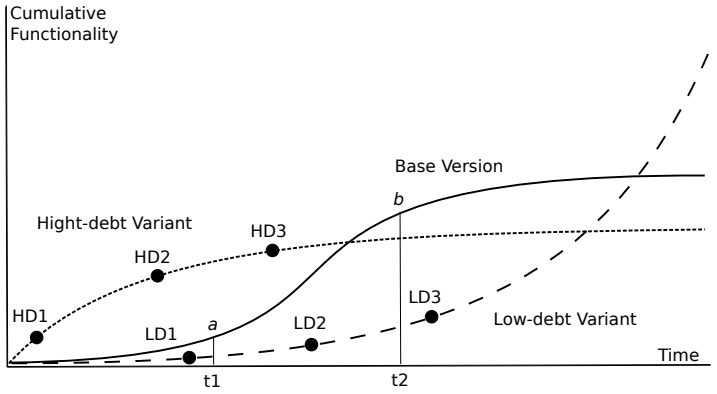


Figure 2.3: Cumulative functionality as a function of time. Product growth trajectories adapted from Ramasubbu and Kemerer (2013)

this inspection is then updated in the correct fields of the TDI.

The estimation can be seen committed on top of the tracking-updated-information, and it derives resource consumption estimates for the captured TDIs'. As the effort required to complete a particular work-package (e.g., delivery of new functionality, optimizing the state of an existing asset by paying of its principal or paying interest in accommodating technical debt in assets' clients) depends on the organization characteristics (the efficiency of involved stakeholders, tools, and development methods for example), this phase requires access to historical data. The data is consulted for previous, similar TDIs and the effort used then to overcome them (Guo and Seaman, 2011; Shull et al., 2013; Guo et al., 2011b). As development environments are always unique, gathering of the historical data is important (Falessi et al., 2013). Field based automatic association of TDIs could be considered to automate this phase, but thus far it has not been attempted, and hence remains a fully manual task (Li et al., 2015).

The technical debt management phase is responsible for decision making to optimize the ROI of the software development's technical debt. Ramasubbu and Kemerer (2013) present the optimization problem in a model and its core is captured in Figure 2.3. On the vertical axis we have cumulative delivered functionality for a software product, while consumed time runs on the horizontal axis. The developing organization is assumed to have a constant amount of resources in use through the observation period. The *high-debt variant* describes a software product growth trajectory which emphasizes quick delivery of functionality for example to quickly break in to a market with a pilot product. However, quick accumulation of functionality places less emphasis on quality while the increasing functionality accumulates complexity for the product. Hence, functionality has an exponential

saturation $F = K(1 - e^{bt})$, where F = added functionality to the base platform, K = functionality's maximum equilibrium value, t = time, and b = functionality growth rate coefficient, curve as more and more resources must be committed to maintaining the technical debt accumulated early on.

The *low-debt variant* accumulates new functionality much slower as more emphasis is put on quality (e.g., refactoring for complexity management). While the start is slow for this software product growth trajectory (and it is hence unsuitable for quickly breaking into a market), software reuse ensures that having low complexity benefits the product in the long run. Large but easily extensible base product allows delivery of new functionality with an exponentially growing $F = e^{bt}$ rate.

Ramasubbu and Kemerer (2013) have identified crucial decision making points for the optimization and they are also present in Figure 2.3. The *HD1* and *LD1* points describe cut-off times for product managers' decisions regarding which trajectory will be taken. Noteworthy is that since the high-debt variant accumulates technical debt much quicker, *HD1* is reached much quicker than *LD1*. *HD2* indicates a point from which the project can get an adequate estimate of the debt obligation that they have taken as main portion of the functionality is delivered and thus the accompanying technical debt has also been accumulated.

At point *LD2* estimates on missed opportunity due to low function delivery can be made, and the trajectory of the product can be further adjusted accordingly. Finally, *HD3* marks the saturation for the high-debt variant. Optimally, if the debt can be written off (i.e., interest probability made zero for all TDIs; see Table 2.2) at this point (e.g., by retiring the product) then the return for this debt investment has been maximized (the difference between the high-debt and the neutral *base platform* variants is the largest). However, if the high-debt variant can not be retired at this point, the debt realizes as inefficient resource use from this point onwards and will reduce or even negate the ROI. Point *LD3* marks the end stage for the low-debt variant as the well built platform enables quick delivery of functionality. As visible from the figure, it is important that the project is able to continue beyond this point to convert its long continued negative ROI into a highly positive one. Thus, we have seen that both the high- and low-debt variants present risks: the high variant leverages debt for quick ROI while the low variant builds on minimal debt to ensure sustainability and long-term ROI. The high-debt variant depends on the product retiring while the low-debt variants counts on the opposite.

The above discussed optimization problem provides a high-level view into software development technical debt management. Decisions on lower levels are required to realize the high level vision (e.g., following or switching between growth trajectories). It becomes apparent that the accumulated Technical Debt List (TDL), representing the whole software development

project's technical debt situation, is vital to the high-level decision making while realization of the management requires propagating the decisions to the TDL's elements, the Technical Debt Instances (TDI). For these, approaches can be adopted from the financial domain, and Seaman et al. (2012) have discussed three such approaches: Simple Cost-Benefit Analysis, Analytic Hierarchy Process, Portfolio Approach, and Options. In the *Simple Cost-Benefit Analysis* for each TDI, two values are calculated the "benefit" and the "cost". The benefit of resolving the TDI is based on the estimate about the TDI's interest (see Table 2.2); that is, if the TDI would be removed then the interest will not be paid and counts as benefit. The effort is calculated based on the TDI's principal (see Table 2.2); that is, in order to get the benefit we much make an effort to remove the principal. Then, the cost-benefit analysis simply chooses those TDIs for management that have the highest benefit related to the lowest cost.

The *analytic hierarchy process* could be considered a more elaborate variant of the previous. Here, a criterion hierarchy, similar to software quality standards (e.g., ISO/IEC 25010:2011 where the quality attribute 'maintainability' considers attributes like 'modifiability' and 'testability') is built, and pair-wise comparisons through the hierarchy are used to produce prioritization weights for the fields of the TDIs. The *Options* method considers each TDI as an investment point where refactoring (i.e., paying the debt) can be used to purchase the option to change these assets in the future. Hence, more likely a module is to change the higher its option value. Finally, the *portfolio* approach is an extension to any of the prioritization methods. The point in this, like in its financial counterpart, is to adjust the contents of the portfolio to pursue a desired ROI with a certain risk. After assessing the TDL, TDIs can be chosen from it to form a desired portfolio.

2.5 Motivations for Further Research

Previous sections have discussed the definition, emergence, and effects of technical debt in order to present how management can be applied for it. These topics were introduced by overgoing existing research from the technical debt and software engineering fields. From this research, we may observe a set of matters that affect efficient implementation of technical debt management: discussed challenges, highlighted future work, or implications derivable from conjoining discussed research. In this section, we review these matters, and we present them as motivations (*M1* –*M5*) for further research to pursue efficient technical debt management.

Having efficient technical debt management in the software development is a multifaceted demand. Firstly, we need to ensure that the management approaches account exhaustively for all technical debt (Motivation *M1*).

Second, the management approaches must be efficient so as to benefit the software development (Motivations *M2–M3*). Finally, the approaches must adapt to the software development environment to enhance work in the particular domain (Motivations *M4–M5*). In the following, a subsection is dedicated for reviewing each motivation in detail. As section contents, the motivation and the possible, inherent mechanisms of the motivation are established via revision of associated literature from the software development and technical debt management research domains.

2.5.1 *M1*: Technical Debt Identification and Estimation are Subjective and Context Dependent

Exhaustive identification of technical debt is an apparent requirement for exhaustive technical debt management. Hence, we examine the premises and current research relating to the first motivation (*M1*) which states that *Technical Debt Identification and Estimation are Subjective and Context Dependent*.

As described in Section 2.1, technical debt, in a software project, corresponds to the cumulative deviation between the software assets' current and optimal states. The software assets are created in a certain technology context. The context defines, with varying degrees of freedom, the allowed objects and their interconnections. The assets are created by following this context definition. It is evident that both, the current state and the optimal state of an asset, are constructable by following the same definition. Further, from the perspective of the technology context, both these states can be considered correct. If an asset would have an erroneous state, then the error caused by this state would induce repairs for it (Bruegge and Dutoit (2004) provides an example of this in describing several feedback loops between defined software assets). For some technology contexts, the semantics of the context induce automatic repairs for the assets. For example, if the bubble-sort algorithm's implementation would be semantically incorrect, the compile-time error would induce repairs for this asset.

If the current state and the optimal state of the software asset are both correct from the perspective of the technology context (e.g., implementation assets follow the syntax of a particular programming language and hence the semantics deliver correct functionality), determination of the current state's sub-optimality can not be based solely on the semantic and syntactic definitions provided by the technology context. Schmid (2013) describes a formal model for technical debt evolution. In this model, technical debt is described as the *"difference between the implementation's evolution cost and the optimal evolution cost of another implementation, which is behaviorally equivalent in all relevant aspects"*. Noting that (*M1.a*) the optimal system is considered a non-reachable and non-pursued ideal that is utilized

only for describing the difference (based on the "good enough" state discussed below), and (*M1.b*) the attained cost figure directly describes the effort required to create a particular implementation in a particular development environment, it can be concluded that technical debt identification requires producing effort estimates about the ideal where subjective input acknowledges and describes the environment conditions.

Hence, technical debt identification is subjective, and a number of researches have called for subjective input. In discussing the use of technical debt data in decision making, Seaman et al. (2012) note that both the principal and effort estimates require subjective input. Falessi et al. (2013) gather requirements for tool supported technical debt management, and in this they note that a balance between expert opinions and automated estimates should be found. Regarding empirical results and knowledge distribution in software environments, Shull et al. (2013) note that a reliable knowledge database enhances technical debt estimates, but even in their absence, an expert opinion should be sought as other alternatives for gathering this information are scarce.

Regarding identification of technical debt, the previous description denied relying solely on context definitions for identifying technical debt. However, we note that the state of the software asset results from the state's implementation and, possibly, from the states of related software assets (see Section 2.2 wherein previous phases of the software life-cycle produce assets that describe implementation of assets in following phases). Hence, identification and estimation of technical debt is context dependent as it is based on the estimate of the optimal state of an asset which can only be described by its implementation context and/or by the implementation contexts of its related assets. Especially, this is noted by Nugroho et al. (2011)'s empirical model of technical debt which derives different repair effort estimates for different technology contexts. Contexts of the related assets, however, lead to a further challenge of being able to understand how the sub-optimality transitions between the contexts. This issue is further discussed by the next motivation.

Closing discussion on this motivation, we identify two issues affecting it. Firstly, the motivation's first mechanism (*the optimal system is considered a non-reachable and non-pursued ideal that is utilized only for describing the difference*) is based on the existence of an "optimal" state. For example Buschmann (2011) and Klinger et al. (2011) discuss a "good enough" state, as it reflects the decision taken in the real world. The issue with the optimal state is that after the good enough state has been reached, pursuing a further state can become highly inefficient. However, the optimal state is used herein as it highlights that there will be technical debt in the product even though it is not acted upon. Finally, McGregor et al. (2012) note in their theoretical work on technical debt aggregation in ecosystems that aggregating technical

debt may compound to become larger than the combination of source debts. Thus, retaining awareness over all technical debt is considered important.

Secondly, the motivation discusses matters as "context dependent". This possibly steers the discussion to identifying independent contexts and their properties. However, the intention is not to establish unique contexts, but, rather, identify all contexts, establish their similarities, and thus allow applicable procedures to be shared among all receptive contexts. Hence, acting on the motivation, one should pursue for minimizing the amount of contexts which is why their characterization is highlighted by Motivation *M5*.

2.5.2 *M2*: Software Development will Associate Software Assets with Each Other

As discussed in Section 2.1, software assets which refer other sub-optimal (i.e., technical debt ridden) assets are probably required to adapt to the sub-optimality (i.e., accumulate further technical debt). Hence, it is vital to technical debt management to understand the possibilities for how assets can refer one another, and thus to understand the ways with which technical debt can propagate in the software development. Motivation *M2* expects that *software development will relate software assets with each other* as there are mechanisms in place that ensure relation in the software development (*M2.a*) through the different phases of its life-cycle and (*M2.b*) through the different contexts present in the phases.

Regarding phases of the software development life-cycle, we identify mechanisms that relate software assets within the same life-cycle phase and between the life-cycle phases. Reuse, the unique characteristics identified for software in the introduction of Chapter 2, is the primary mechanism (*M2.a.i*) relating software assets with each other in the same software life-cycle phase. Most prominent example of this is the reuse of assets in the software implementation phase where semantics dictate reuse to be the sole mechanism for asset generation. For example, the bubble-sort algorithm is constructed through references to list structures, list operators, and value comparison operators. Reuse manifests in most software development life-cycle phases as the technology contexts generally provide the phase with semantics that enable it. Most prominently, the Unified Modelling Language (UML), which is a widely used technique in the design phase, declares several reuse mechanisms for the language's elements (c.f., 'Behavior' Section 13.2 of the UML 2.5[10] specification).

Regarding relation of software assets between phases of the software development life-cycle, the relation can exist between (*M2.a.ii*) a phase and another phase that follows it or (*M2.a.iii*) a phase and another phases that

[10] `http://www.omg.org/spec/UML/2.5/`

precedes it. Reviewing Figure 2.1 we see that realization of the software development life-cycle relies on mechanism *M2.a.ii*, as assets in later phases are created by referring assets produced by the previous phases. However, for example, iterative and incremental software development methods state the same for mechanism *M2.a.iii* as well when previous versions are used to gather feedback on and to modify existing requirements and designs (Larman and Basili, 2003; Royce, 1970; Davis et al., 1988). The Scrum method also provides an example of this mechanism as it requires a Sprint to deliver a functioning sub-product for client review (Schwaber and Beedle, 2002). Here, the implementation assets (from the implementation phase) deliver functionality and the client review targets them. The review however produces changes to the requirements (from the requirements elicitation phase) in order to be fed back into implementation through analysis and design.

As there a number of different stakeholders involved in the software development (see Table 2.1), and most stakeholders work with and communicate about only a certain sub-set of software assets, the demand for updating the non-planned changes from later phases of the software life-cycle into the previous phases can be seen justified. For example, designer or client representative stakeholders, who work on the abstract design assets, are dependent on the developer stakeholders updating the design assets to reflect any deviating interpretations of the designs in the implementation assets (Bruegge and Dutoit, 2004).

There are multiple different technology contexts within which software assets, from the software development life-cycle, are defined and used. Hence, we identify mechanisms that relate assets (*M2.b.i*) within the same technology context and (*M2.b.ii*) between technology contexts in order to assure that there are no areas from the life-cycle that are disconnected from the whole. For mechanism *M2.b.i*, we note that a technology context limits to a particular software life-cycle phase, the context is referred when producing assets in the phase, and, hence, mechanism *M2.b.i* corresponds to mechanism *M2.a.i*. Inter-technology-context relation (i.e., mechanism *M2.b.ii*) is a particularly interesting phenomenon, as no formal semantics are available to describe, the very apparent, relations here. The semantics of a particular technology context (e.g., a programming language) describe relations for assets in this context, but these assets commonly relate to assets in other contexts that reside in (*M2.b.ii.*α) the same software life-cycle phase or *M2.b.ii.*β) in another life-cycle phase. For mechanism *M2.b.ii.*α, an example of a relation between assets that reside in the same life-cycle phase but in different contexts is a database connection implementation. Here, both assets, the connection implementation and the database implementation, are in the same life-cycle phase (the implementation phase from Figure 2.1). However, the connection implementation follows a technology context (e.g., the Python programming language context) that is different from the

database implementation (e.g., the MySQL database definition context). The relation between these assets is very apparent, but the semantics of neither describe it.

However, and as an instance of mechanism *M2.b.ii.β*, it is probable that a design asset has described the database connection for implementation. For example, a UML sub-system decomposition describes that there exists a sub-system responsible for the database connection and it uses the interface of the database sub-system. Two inter-context inter-phase relations can be identified. Both are from the design life-cycle phase to the implementation phase. The database connection sub-system description is a relation between the UML context and the Python programming language context while the database sub-system description relates the UML context to the MySQL database definition context.

Reviewing the six aforedescribed mechanisms which describe how software assets relate with each other independent from the software life-cycle and the technology context, indicates that active software development will relate software assets with each other (*M2*). This motivation is further backed by the notion that an asset can be seen to produce value only when it is referred: only the reference (by another asset or an actor outside the system) will utilize the asset.

A number of research outcomes have discussed the phenomenon of software asset relation as part of technical debt management. Klinger et al. (2011) note in their enterprise case study that issues in the software architecture can affect an entire portfolio of software products under it. Similarly, Wiklund et al. (2012) note in their case study that stakeholder assumptions play a role in test design, where infrastructure features (e.g., hardware delay for time-sensitive application tests) are implicitly taken into account. Further, technical debt solutions that identify debt for a particular life-cycle phase can execute the measurements for it in another life-cycle phase. Li et al. (2014) investigate the use of modularity metrics for identifying architectural technical debt. Here, identification of technical debt in the design phase requires measurement in the implementation phase. Additionally, they note that the cost to fix architectural violations depend on the context and the project in question. Marinescu (2012) also propose measuring design debt via identification of design flaws by executing measurements in the implementation phase.

An issue can be noted for this motivation in its discussion about relating software assets. As we have previously noted, technical debt is not limited to static assets: for example, socio-technical connections exist between individuals (Tamburri et al., 2013). As such the motivation is limited in its view of what relations a software development may form, and thus will lead to a limited discussion on a possible solution. However, extending beyond assets and leaving the static domain poses challenges that nullify other ef-

forts described herein. Firstly, if the static presumption does not hold, then rule based manipulation can not be applied as there is no certainty over the applicability of the rules when the target is dynamic. Second, importance of the subjective opinion has been discussed (c.f., identifying technical debt items in Section 2.4). This approach, however, poses a severe risk if extended beyond software assets as personal opinions and other underlying factors come to affect a person's judgement (c.f., Walfish et al. (2012)).

2.5.3 *M3*: Software Development Imposes Update Frequency Demands for the Technical Debt Information

Software development comprises the creation of software assets (e.g., a requirements document, a system design, or an implementation element) which either enable the creation of further such assets, or running of the assets on selected hardware in order to realize the asset's functionality. Software assets are thus closely inter-connected (see *M2*), and a change in one asset is likely to impose change on the related assets as well. These assets can reside in the same or differing phases of the software development life-cycle and in the same or differing technology contexts.

Technical debt was identified in *M1*, following Schmid (2013) formalization, to be the deviation of a software asset's state from its optimum during the asset's evolution. The evolution corresponds to changes in the software asset; possibly induced by related assets changing as described above (see *M2*). Hence, to ensure information about technical debt in a particular software asset is current, a similar identification process must be completed every time the software asset's state changes. Thus, Motivation *M3* expects that *software development imposes update frequency demands for the technical debt information.*

We can observe the software development to impose two frequency demands onto the technical debt information: (*M3.a*) the information production frequency and (*M3.b*) the information consumption frequency. The information production frequency is higher and bases on the aforedescribed software asset state alteration rates. The information consumption rate is lower and bases on the software development project's management strategy.

The aforedescribed requirement to re-identify technical debt as the software asset's state changes is in the core of the high update frequency demand. If we map the interconnectivity of assets through the software development life-cycle phases and the contexts present in them, combinatorial explosion takes place. This is due to software assets' capability to both refer back to previous assets and the reuse mechanism's capability to superlinearly accumulate relations (i.e., all software assets have the ability to directly reference multiple—or in some cases all—software assets from the software development project, and the referred assets can in turn do the

same to accumulate indirect relations in a fashion that is super-linearly comparable to the number of assets present in the project). This indicates that, especially mature, software assets accumulate a tremendous amount of relations, all of which can trigger state changes in the asset. Hence, the asset-based update frequency demand is considerable.

On the other hand, the consumption rate of technical debt information is much slower than its production rate. Technical debt information is consumed as part of technical debt management efforts (see Section 2.4) which function as part of the management strategy chosen for the particular software development (Bruegge and Dutoit, 2004; ISO/IEC 12207:2008). The management strategy can be seen to consist of software development method (*M3.b.i*) independent and (*M3.b.ii*) dependent strategies. From independent strategies, we can identify that stakeholders manage software assets when ever they work with them (e.g., a client representative manages a use case documentation when he/she access the document to make an update to its description (Bruegge and Dutoit, 2004; Zhu et al., 2006)). Hence, for this, the technical debt information update frequency demand is directly derivable from the access rate of particular assets. From dependent strategies, we can observe that modern iterative and incremental (see Section 2.2) methods impose update frequency demands based on iteration management needs. Hence, we can identify that for example the Scrum method, which defines the *Daily Standup*, *Iteration Review*, and *Iteration Retrospective* management practices, requires updates to the technical debt information once per day to satisfy the needs of the Daily Standup while the Iteration Review and Iteration Retrospective require updates once every iteration length (Schwaber and Beedle, 2002). Further, the granularity of the information update varies also; manifesting for example as technology context limits and estimation accuracy. The granularity is at its finest when a stakeholder requires information on a particular asset, and it may be at its coarsest when several stakeholder groups meet for the Iteration Retrospective to get a general sense of the project's state.

Regarding update frequencies to the captured technical debt information, Seaman et al. (2012) note for the technical debt management framework that availability and currency of this information is vital to the success of the framework's application. Further, the technical debt information domain appears to differ from other management information domains, as Griffith et al. (2014) did observe no relationship between software quality models and technical debt estimation approaches.

Finally, an issue is noted for this motivation. While the motivation is capable of identifying a number of demands from software development that affect technical debt information processing. One should, however, also consider the opposite: can technical debt information pose demands on to software development? Characterization efforts of technical debt instances

and software environments in Research Question *RQ3* demonstrate a possible way of overcoming the issue. In case such demands could be exposed and their use facilitated, the outcome could lead to extending existing technical debt management approaches with reactive approaches. Reactive management culture is common to start-ups (Paternoster et al., 2014) and already accommodated in many agile methods (Schwaber and Beedle, 2002).

2.5.4 *M4*: Organizations and Projects Have Unique Technical Debt Information Needs

A software development life-cycle is realized by the hosting project which again exists as part of an organization (Bruegge and Dutoit, 2004). The project realizes the life-cycle via the chosen software development methods (see Section 2.2). The methods define practices and processes which manage the work, the produced assets, and the stakeholders involved in the life-cycle. The project adopts the methods while taking into account characteristics of its environment (Dybå and Dingsøyr, 2008). As this makes the project's software development life-cycle unique on all aforementioned accounts, Motivation *M4* sees *organizations and projects [to] have unique technical debt information needs*.

The chosen management strategy shapes the life-cycle's technical debt information needs in two notable ways. Firstly, (*M4.a*) it affects the update frequencies required from the information (this matter was discussed as mechanism *M3.b.ii*). Second, (*M4.b*) the management strategy has put in place a documentation structure that the software development project abides to. The produced technical debt information needs to consider this: the documentation structure distributes information within the software development project and hence it is in the core of the project's decision making. For example, the Scrum method's backlog artifacts: the *Project Backlog* documents the tasks for the entire life-cycle and the *Iteration Backlog* takes a sub-set of these tasks for the next iteration to focus on. Here, if integrating technical debt management for existing software development life-cycles, inputting technical debt related tasks into the backlogs is up for consideration. The inverse of this is also of interest, as technical debt management approaches could find and subject technical debt related tasks from the backlogs.

The management strategy is also influenced by the executive vision which reflects the organization's aims. This is visible in (*M4.c*) the estimation dependability required from the technical debt information. We can see that ad-hoc and small-scale software developments, due to their agility, both produce and can work on lower estimation dependability. As majority of the available resources are directed towards end-user value production, it is common that infrastructural process have lower priority (Sutton, 2000; Fayad

et al., 2000). Hence, produced estimates allow for low dependability as the development's agility enables making quick corrections if the estimates do not hold. Further, vaguer estimates require less work and thus can be more efficiently produced. On the other end of the dependability spectrum are generally large-scale and more bureaucratic software development projects. They require higher dependability as estimates are processed further and input into planning for different stakeholder groups (Zmud, 1980). As this increases the estimate's value regardless of how refined it is, it is desirable to increase the estimate's dependability. Noteworthy is also that prolonged estimate handling is problematic, as per Motivation *M2*, the information update frequency diminishes the static estimate's value as time progresses.

The technical debt information need is also affected by (*M4.d*) the advancement strategy chosen for the software development (see Product Growth Trajectories in Figure 2.3). Software developments which pursue low-time-to-market (e.g., to trial a new product) may look for technical debt information that optimizes functionality output. Safety-critical and robust developments on the other hand may look for technical debt information that supports risk management. Especially, when the resources available for technical debt information production are limited, it can be expected that emphases on the afore-identified needs will further contribute to the software developments' unique technical debt needs. Falessi et al. (2013) report on their experiences in dealing with technical debt in a CMMI (Capability Maturity Model Integration) maturity level 5 company. They note several practical matters to affect the company's unique technical debt information needs, such as working with different stakeholder groups in order to understand the trade-offs required in quality, cost, time-to-market, and future system maintenance and improvement. From Morgenthaler et al. (2012) description of technical debt management at Google, we can note that the organization requires unique technical deb types (i.e., dependency and visibility) and information attributes (i.e., zombie code and dead flags) to enable efficient technical debt management. Shull et al. (2013) note in their work that technical debt concepts are context-specific, and information to determine if projects are similar enough to transfer technical debt knowledge between them is scarce.

2.5.5 *M5*: Technical Debt Instances Are Polymorphic Assets for Management

Motivation *M2* listed the multiple mechanisms that were identified for software development with which it can relate software assets together. Its capabilities extended the relation-forming beyond single software development life-cycle phases and their technology contexts. As technical debt exists as the sub-optimal state of software assets, and referring assets may be sub-

jected to adhering to the sub-optimality, it seems evident that the previous mechanisms evolve technical debt instances in a similar life-cycle-phase-and-technology-context-exceeding manner. Hence, Motivation *M5* sees *technical debt instances [to be] polymorphic assets for management.*

We identify two polymorphic characteristics of a technical debt instance as well as issues that these characteristics can be seen to cause on the instance's technical debt identification, estimation, and management. (*M5.a*) Technical debt instances transfer and span from software life-cycle phase and technology context to others (following the description provided in the previous paragraph; referring *M2* mechanisms). Regarding issues emergent from this characteristic, (*M5.a.i*) the stakeholder can become detached from the instance. If the instance's context or life-cycle phase changes unbeknownst to the organization, and if the stakeholders in these areas work at least partially independent from one another, there is a clear possibility that the subjective knowledge already accumulated regarding this instance is lost or not fully utilized. This hinders the instance's management efforts as subjective knowledge is required for it as per Motivation *M1*.

Further, as the technical debt instances cross software development life-cycle phases and technology contexts, (*M5.a.ii*) the effects of the instance change. Sub-optimal asset state in another context will have different effects as the effort is context dependent (as described for Motivation *M1* following Schmid (2013) work). Noteworthy is that there is at least a hypothetical chance of a previously low-impact technical debt instance becoming a high-impact show-stopper when it propagates to a critical asset. The final issue for this characteristic is (*M5.a.iii*) technical debt management method applicability. Motivation *M1* discussed that technical debt identification and estimation is technology context dependent. Technical debt management can be seen to have context independent parts, but implementation of a management decision will require further context knowledge (stakeholder executes the management decision on a particular software asset which is defined by a particular context; e.g., a modification to an implementation element can require knowledge of a particular programming language). Hence, as the technology context of a technical debt instance changes, it can be seen to also affect the applicability of different context specific methods applied for it. For example, a tool which is used to track instances in a particular context can assume a newly found sub-optimality to encompass a new instance while, in fact, the instance already has an evolution history. Further, if the instance then leaves this context, should the tracking tool release tracking resources from it and at which point? The method applicability issue appears to capture several concerns regarding technical debt management efficiency.

The second polymorphic characteristic of technical debt instances is (*M5.b*) merging. As per Motivation *M2*, software assets relate with one

another, and technical debt corresponds to the sub-optimal state of a software asset, it is apparent that the sub-optimal state can be the result of cumulative adaptation to sub-optimal states in referred assets. A foreseeable issue is that (*M5.b.i*) it obfuscates distinguishing technical debt instances. If an instance merges with others in a particular software asset, then the software asset acts as an obfuscator (i.e., a "black box" described by Myers et al. (2011) for software testing): the asset may cause adaptations in its referrers, but it is unknown which technical debt instance in the asset contributes, and to what degree, in these adaptations. This can be expected to hinder for example producing technical debt information updates (see Section 2.5.3). Further, relating closely to issue *M5.a.iii* in the previous paragraph, as merging and splitting becomes obfuscated for the technical debt instances, the technical debt management methods become stressed: a particular software asset may be subjected to over management, if several technical debt instances are tracked to it. This issue further indicates possible concerns for management efficiency.

Reviewing the previously described polymorphic characteristics for technical debt instances and the potential issues emergent from the characteristics, it seems evident that technical debt instances are not simple, monomorphic items which could be subjected to the same management methods throughout their entire life-cycle. Research has also identified this, as Zazworka et al. (2013) report that in their interview of different stakeholder groups, all but one identified differing technical debt instances. In Tom et al. (2013) multivocal literature review on technical debt management, they noted that a single technical debt instance can be classified in more than one form (i.e., strategic, tactical, incremental, and inadvertent). Finally, we re-note the hypothesised compound property for technical debt (discussed in Section 2.5.1 from the research of McGregor et al. (2012)) where it is possible that the compounded technical debt is greater (i.e., it may have a larger effect) than the sum of its parts.

A matter which should be discussed in relation to this motivation—applying to Motivation *M4* as well—are the properties that are unique to a particular instance of technical debt. While it is more efficient, and thus lucrative, to search for common properties, it should also be established what is the role of unique technical debt properties in terms of implementing management for the individual instances. Regarding the notion of impact of unique instances, it should also be established if commonly applicable management approaches can occupy all resources intended for technical debt management or should they also account and thus spare resources for the management of unique instances?

M1 Technical Debt Identification and Estimation are Subjective and Context Dependent

- a) the optimal system is considered a non-reachable and non-pursued ideal that is utilized only for describing the difference
- b) the cost figure describes the effort required to create a particular implementation in a particular development environment

M2 Software Development will Associate Software Assets with Each Other

- a) through the different phases of its life-cycle
  - a.i) relating software assets with each other in the same software life-cycle phase
  - a.ii) a phase and another phase that follows it
  - a.iii) a phase and another phases that precedes it
- b) through the different contexts present in the phases
  - b.i) within the same technology context
  - b.ii) between technology contexts
    - b.ii.α) the same software life-cycle phase
    - b.ii.β) in another life-cycle phase

M3 Software Development Imposes Update Frequency Demands for the Technical Debt Information

- a) the information production frequency
- b) the information consumption frequency
  - b.i) software development method independent
  - b.ii) software development method dependent

M4 Organizations and Projects Have Unique Technical Debt Information Needs

- a) it affects the update frequencies required from the information
- b) the management strategy has put in place documentation structure for the software development life-cycle which the technical debt information needs to consider
- c) this is visible in the estimation dependability required from the technical debt information
- d) the advancement strategy chosen for the software development

M5 Technical Debt Instances Are Polymorphic Assets for Management

- a) technical debt instances transfer and span from software life-cycle phase and technology context to others
  - a.i) the stakeholder can become detached from the instance
  - a.ii) the effects of the instance change
  - a.iii) technical debt management method applicability
- b) technical debt instances merge
  - b.i) it obfuscates distinguishing technical debt instances

Listing 2.4: Motivations with their identified mechanisms

Chapter 3

Research Description

This chapter describes the research committed under the thesis's subject *Technical Debt in Software Development – Examining Premises and Overcoming Implementation for Efficient Management*. The description is provided in two parts. First part introduces the thesis's research questions (*RQ1–RQ3*) in Section 3.1. The research questions are introduced on top of the motivations (*M1–M5*) that were derived from existing research in Section 2.5. A brief description of research methodology in Section 3.2 acts as an intermediary. The second, final, part is Section 3.3. It describes the research approach taken in the attached original publications (*P1–P7*) to overcome the set research questions. Figures adjoin the respective sections to describe how the motivations relate to the research questions (see Figure 3.1), how the publications serve the research questions (see Figure 3.2), and, in the following chapter, how individual contributions from the publications map to the previous (see Table 4.1).

3.1 Research Questions

This section describes three research questions which organize work to meet the dissertation's target of *Examining Premises and Overcoming Implementation for Efficient Management* of technical debt. In Section 2.5 we described a set of matters emergent from existing research that we introduced as motivations for further research into efficient technical debt management. As such, the research questions in this section are built by acknowledging the matters pointed out by these motivations. Figure 3.1 depicts this acknowledgement and it can be described as follows:

- Research Question *RQ1* builds a tool for subjective and context specific technical debt identification to address Motivation *M1*

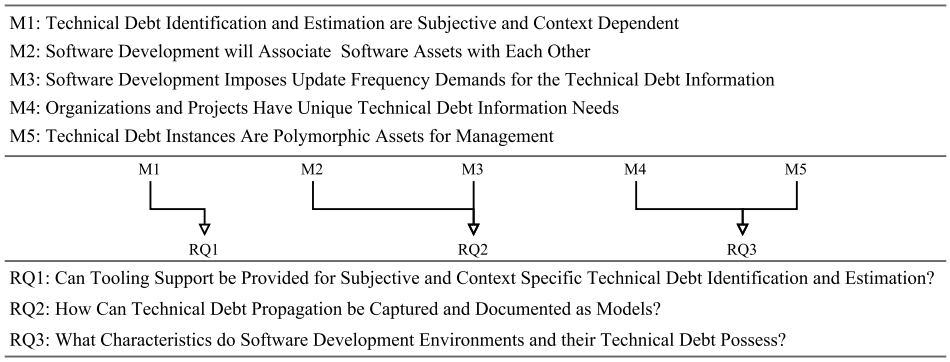


Figure 3.1: The thesis Motivations *M1* through *M5* mapped to its Research Questions *RQ1* through *RQ3*

- Research Question *RQ2* captures technical debt propagation in models to address Motivations *M2* and *M3*. Integration of the models into Research Question *RQ1*'s tool is also pursued to allow the models' automatic assessment to increase the efficiency of technical debt information maintenance

- Research Question *RQ3* surveys existing software development environments and technical debt instances present in them in order to better understand and adapt the contributions of the previous research questions to them whilst addressing Motivations *M4* and *M5*

Whilst addressing the dissertation's target broadly, the core of the research questions' can be described as pursuing *efficient technical debt management with a tailor-integrated (RQ3) technical debt identification and estimation tool and process (RQ1) that supports propagation model based automatic information maintenance (RQ2).*

3.1.1 *RQ1*: Can Tooling Support be Provided for Subjective and Context Specific Technical Debt Identification and Estimation?

Motivation *M1* discussed that technical debt identification and estimation is subjective and context dependent. In the case of technical debt identification the subjective input is limited to the stakeholder acknowledging that there is a technical debt instance (possibly based on already refined information e.g., from static code analysis) and production of information in the instance's documentation. The estimation process requires the participant to acknowledge the particular development environment and to assess the technical debt instance as a part of it to complete the instance's estimate.

Several works from technical debt research acknowledge the subjective component. Guo et al. (2011b) note in their case-study on technical debt tracking that in order to simulate a technical debt instance at time t, one must derive the *InterestAmount(t)* and the *InterestProbability(t)*. And it is stated that for both, in lack of a better method, expert knowledge is required. Ho and Ruhe (2014) come to a similar conclusion in using the Goal-Question-Metric (GQM) (Basili, 1992) approach to making technical debt aware when-to-release decisions. Here, GQM's decomposition to metrics involves quality attributes (e.g., reliability from ISO/IEC 25010:2011), the valuation of which is indicated to require input on stakeholders' satisfaction and weighing based on expert opinion. For the previously referred empirical model of technical debt (Nugroho et al., 2011), the use of expert judgement in determining (i.e., estimating) the repair effort of a technical debt instance is identified as one of the challenges. Finally, Marinescu (2012) notes that in assessing (i.e., estimating) technical debt through design flaws, the *Flaw Impact Score* (FIS) is gathered through expert opinions.

Reviewing the aforedescribed work, we can note that the subjective component plays an import role for the identification and estimation processes of technical debt. However, and likely due to the differing focus of the previous works, we can not identify these or other contributions to actually describe any methods through which this subjective input (i.e., expert opinion) could be systematically and efficiently gathered. As it was noted that the subjective input is required in only very limited parts of the processes (Ho and Ruhe, 2014; Nugroho et al., 2011), meaning that rest of the processes completion can be achieved through other means (c.f., Eisenberg (2012) for a threshold-based automation approach), the first Research Question *RQ1* asks *Can Tooling Support be Provided for Subjective and Context Specific Technical Debt Identification and Estimation?*

The tool pursued by *RQ1* would enhance the efficiency of the technical debt identification and estimation processes if it would be able to complete the non-subjective parts of the processes through other means. We may identify these to be for example the parts of the technical debt items description (see Table 2.2) that can be automatically derived (e.g., the time, the author, and—to a degree—the location). Further, the tool should take into account and adapt to the technology context in which it is used. As *M1* reviewed, technical debt is identified as software assets' state deviations, and the deviation assessments are context-bound. Hence, in identifying and providing estimates for the technical debt instances, the tool should allow for and enable the production of context-specific information. If this is not achieved, the accuracy of produced descriptions, arguably, suffers.

3.1.2 *RQ2*: How Can Technical Debt Propagation be Captured and Documented as Models?

Motivation *M2* noted the many mechanisms with which software assets can relate with one another. Looking at the number of possible relations intra- and inter-software development life-cycle phases and technology contexts, it is evident that many of the relations are yet to be covered while certain groups of relations closely represent each other. It is important to discover the remaining relations and to make them explicit: assessment of all possible relations is required to fully value in technical debt instances for management. Further, in cases where the number of relations disallows exhaustive assessment, it is important to have knowledge of the most propagation-capable and hence the most effect generating relations.

Regarding groups of similar relations, it is again vital to acknowledge these groups as this will allow transferring already acquired knowledge between the group's members. If for example an effort estimate can be generated for a particular group member (i.e., a relation), then, with limitations of the generalization applying, the same estimate can be related with other members of the group. This is highly preferred from the point of view of efficiency, as domain knowledge gets automatically distributed.

Research Question *RQ2* asks *How Can Technical Debt Propagation be Captured and Documented as Models?* in order to satisfy the aforementioned need of discovering software asset relations and enabling grouping between similar relations. The need to discover relations can be seen requested by, for example, Marinescu (2012), Eisenberg (2012), Li et al. (2014), and Tamburri et al. (2013), cited in *M2*, that examine or suggest technical debt management solutions based on assessment of the relations. Further, the request for grouping relations to enable efficient knowledge transfer is justified in observing that particular domains are often conjoined in a higher context level. For example, the object-oriented programming paradigm conjoins the Java and Python languages. This enables knowledge transfer from one language's relations to another. Given that similarity between the relations (i.e., the limitations of knowledge transfer) of both languages has been established beforehand; through language definitions in this case.

The research question will produce a method which is capable of capturing technical debt propagation from existing and simulated software projects. This should satisfy the discovery need, as the method's execution over projects discovers new ways for technical debt to propagate (i.e., software relations of interest). Further, in order to capture propagation into models, the method needs to group similar software asset relations with one another (as *M2* discussed this as the means for technical debt propagation). The produced models can then be used to infer, from an identified root asset, the effects of technical debt onto other, related assets.

Second, capturing technical debt propagation in models provides a valuable opportunity for answering the technical debt information update frequency demands imposed by software development (see Motivation *M3*). Capturing technical debt propagation in a model requires describing a relation between software assets which is responsible for realizing sub-optimal state deviation in one of the assets. In this relation, there is a source asset, a destination asset, and a description of a process which realizes the deviation. As this corresponds to capturing semantics (i.e., defining behaviour between an input and output pair (Slonneger and Kurtz, 1995)), we may ease the technical debt information update process via operationalization of the relation's captured semantics. The operationalization can be seen to range from fully automated to assisted. The former expects semantics for which an interpreter already exists (i.e., a programming language) while the latter is limited to demonstrating the destination asset whilst requiring manual input to derive the actual information update.

Existing efforts to model the propagation of technical debt are limited. We can only identify the work by McGregor et al. (2012) which hypothesizes on two mechanisms for technical debt to propagate within components of a software ecosystem. The first mechanism states that *"technical debt for a newly created asset is the sum of the technical debt incurred by the decisions during development of the asset and some amount based on the quality of the assets integrated into its implementation"*. They further note that layers of implementation may diminish the amount of aggregated debt. Analyzing the mechanism, we may note that the decisions closely reflect the sub-optimal state accumulated for a software asset (see Section 2.3) while the quality of the integrated assets is similar to the previously discussed mechanism of adhering to sub-optimalities (see Section 2.5.1). The second mechanism states that *"technical debt of an asset is not directly incurred by integrating an asset in object code form, but there is an indirect effect on the user of the asset"*. An example is used where lacking documentation makes integration of the asset more cumbersome. Again, we may note that this mechanism closely captures what Motivation *M2* discussed as mechanism *M2.b.ii* (i.e., relation of software assets which reside in differing technology contexts).

3.1.3 *RQ3*: What Characteristics do Software Development Environments and their Technical Debt Possess?

The technical debt management tool and the propagation models produced by Research Questions *RQ1* and *RQ2* are quantitative results. In addition to the exhaustive meeting of the results being extremely cumbersome (combinatorial explosion discussed in Motivation *M2* increasing the number of possible propagation models to explore), they will produce value only

when applied to, pre-existing, software engineering environments. Hence, in order prioritize the work in and to adapt the contributions of the previous research questions, Research Question *RQ3* pursues observing and understanding the empirical, and especially qualitative, side of software engineering. Concretely, we do this by systematically answering the call to *characterize 1) software development environments and 2) their technical debt.*

First part of the characterization targets software development environments. Contributions discussed in *RQ1* and *RQ2* will be integrated into these environments. There are various software development environments (e.g., varying methods and organization structures) (Dybå and Dingsøyr, 2008; Licorish et al., 2016), and vast majority of them are pre-existing (i.e., excluding newly formed software development organizations)[1]. Due to this, Motivation *M4* discussed that many environments can be expected to have unique technical debt information needs. As such, they need to be characterized in order to establish an interface through which the contributions are 1) integrated into the software development environments and 2) their performance can be monitored.

For integration, the interface must establish wherein information relating to technical debt management is available (i.e., produced) and where it is required (i.e., consumed). Generally, both of these boil down to specific software development practices and/or assets present in the environment in question. The contributions successful integration equals to being able to enhance the software organizations' technical debt information content or its distribution (as discussed in Motivation *M4*).

Monitoring the performance equals to establishing that technical debt is being efficiently managed. Hence, monitoring is implemented by tracking the development and value generation speed of particular organizations throughout integration of new or enhancement of existing technical debt management approaches. Enhanced management should allow the organizations to increase the debt's ROI which again should lead to increased value generation speed. It is also important to note that the interface implicitly establishes where technical debt management can not be applied, or where the contributions are incapable of addressing management, as the inversion of the integration interface will encompass these areas.

Second part of the characterization targets the technical debt existing in the software development environments. Existing software development environments also host the mechanisms emergent to new technical debt instances in addition to the already spawned instances. Motivation *M5* discussed technical debt as a polymorphic asset for management (Zazworka

[1]For example, about variety in the use of agile methods, the industry driven *State of Agile Report* `http://stateofagile.versionone.com/`

et al., 2013; Tom et al., 2013). Several issues were seen to arise as the instances travel and span over different phases of the software development life-cycle and different technology contexts (Schmid, 2013; McGregor et al., 2012). Issues arising from the technical debt instances being transferred and spanned from software life-cycle phase and technology context to others (*M5.a: "technical debt instances transfer and span from software life-cycle phase and technology context to others"*) can only be characterized based on empirical evidence as their realization and variations are dependent on the software development environment in question. Similarly, technical debt instances merging and splitting (*M5.b: "technical debt instances merge"*) is another matter that has to be considered. While the propagation models are able to capture certain mechanism (i.e., pre-existing semantics explain adaptations), there are also mechanisms with qualitative attributes (e.g., a software developer's ability to conceptualize when he/she interprets a design asset to convert it into an implementation asset, as noted for *M2.b.ii.β: "software development will associate software assets between technology contexts in different software development life-cycle phases"*).

Characterization of existing technical debt instances is thus vital to successfully integrating the contributions into the software development environment: we must be able to understand where technical debt resides, which life-cycle phases it affects, and how different instances behave in differing environments. Further, as Motivation *M2* discussed technical debt channels to be prone to a combinatorial explosion in channel variations, limited development resources require that an effort is made to prioritize the most important technical debt propagations for management.

3.2 Research Methodology

The section in question provides a general review into conducting research. This takes into account the generally applicable reasoning approaches, research methods, and ways and characteristics of implementing research. Section 3.3 will then describe and contrast individual contributions of the thesis against this review.

Reasoning approaches available for a particular research undertaking can be generally considered to be either deductive or inductive (Trochim, 2016). *Deductive reasoning* is hypothesis testing. Here, an initial theory exist and a hypothesis is generated to test—usually a part of—the theory. This is followed by a research design which produces observations that can be used to either confirm or decline the hypothesis. In most cases, and especially when several variables co-exist, multiple hypotheses are required to prove or adjust and prove the theory. *Inductive reasoning* approach can be considered the inverse of deductive, and it generates new hypotheses.

Herein, the world is observed as exhaustively as possible. Following research tries to then identify patterns from the observations to build a tentative hypothesis. Further observations and pattern identification will then follow in order to provide further evidence for the hypothesis or its adjustment so as to contribute a theory. (Trochim, 2016)

Research methods in software engineering are discussed by Basili (1993) as the four different models for conducting research. They are the scientific, engineering, empirical, and analytical methods as later revised, amongst others, by Wohlin et al. (2012). The *scientific method* observes the world in order to build a model or a theory. Further observations are then intended to provide evidence for or against the initially conceived system; to validate the hypothesis. This method is close to pure inductive reasoning. The *engineering method* follows the previous, by studying existing solutions. The aim is to derive improvements for the solutions, assess if their implementation is successful, and halt when no further enhancements can be derived. Basili argues that as an "evolutionary improvement oriented approach" (Basili, 1993), the engineering method is quite suitable for assessing existing software processes, products, et cetera to determine if new tools or other introduced enhancements improve them.

Moving to deductive reasoning, rather than observing the existing, the *empirical method* starts by proposing a model. This can be a completely new way of producing software for example. A requirement of the empirical method is that the proposed model needs to be accompanied by empirical validation procedures, either quantitative or qualitative, that can be executed in order to measure and determine if the model has actually produced an improvement. Finally, as the counterpart to the empirical method for deductive reasoning, the *analytical method* proposes a theory with formal premises. Introduced by Basili (1993) as the mathematical method, derives results to prove or falsify the theory directly from the formal premises. Linkage to the observed world, if possible, can be provided via comparing the theoretical results to empirical observations. Using the bubble sort algorithm, the thesis's running example, as a target for the analytical method: it can be proven that the algorithm's complexity is of magnitude $O(n^2)$ by considering the commands forming the algorithm as mathematical objects. We may then tie the demonstrated complexity to the observable world, by producing empirical measures of the algorithms performance that show a similar dependency to exist between the algorithm's input and average or worst case performance.

Research approaches are used to implement research methods. Reviewing the previous methods, we may note that the analytical method would be the preferred way of conducting research as the formal premises allow us to prove that the theory holds exhaustively. However, most systems—generally all structures of the observable world—are too complex to

be modelled as mathematical objects which requires us to fall back to the other research methods and to empirical evidence gathering. There are several different implementations of this, and literature generally discusses four different approaches: case study, experiment, survey, and action research (Runeson and Höst, 2009; Wohlin et al., 2012).

The case study examines "a contemporary phenomenon in its original context" (Runeson and Höst, 2009). There are several empirical research settings in which it is difficult to fully distinguish or to completely separate between the research subject and the context, or the environment, surrounding it. Hence, there is less control over the examined phenomena and the researcher must settle for monitoring an existing setting in order to gather evidence for analysis (Wohlin et al., 2012). The level of control in case studies disallows exhaustive specification of variables (which is the case for experiments) that would allow identification of variable relations as statistical significances. Runeson and Höst (2009), however, argue that when proper research methodology practices are applied and the definition of knowledge is extended beyond identification of statistical significances, the case study is a prominent research tool. In many a case in software engineering research, for example in software development practice research where the practices function as part of inseparable environment, the case study, possible together with the survey, is the only applicable research approach.

The experiment research approach is on the other end of the control spectrum. As the experiment is about "measuring the effects of manipulating one variable on another variable" (Runeson and Höst, 2009; Robson, 2002), it is a prerequisite of the approach that all variables are identified, fixed, and controlled throughout the entire experiment process. Hence, the setting is often called a laboratory environment. Moving back on the control axis, the quasi-experiment describes a variation of this approaches wherein it is not possible to randomly assign subjects to treatments (Wohlin et al., 2012). The aim of the experiment analysis is to demonstrate a statistical significance between the variables; controlled and uncontrolled. Establishing such a relation in a laboratory environment indicates that no other factor than manipulation of the controlled variable can cause the observed change in the uncontrolled variable; the independent and the dependent variables respectively (Wohlin et al., 2012).

Fully releasing control, *the survey* research approach investigates, often in retrospect, how a certain process, tool, or some other target has existed for a particular population (Wohlin et al., 2012; Runeson and Höst, 2009). Surveys are highly sensitive research tools, and several matters affect their quality (Stavru, 2014). Most notably, a survey is generally executed on a sample which is taken from the population. As such, the sampling method must ensure that the sample represents the whole population well.

Table 3.1: Research method, approach, and characteristics for the included Publications *P1* through *P7*

			Characteristics	
Paper	**Method**	**Approach**	Objective	Data
P1	Scientific	Action Res.	Exploratory	Qualitative
P2	Engineering	Survey*	Exploratory	Qualitative
P3	Scientific	Case Study	Descriptive	Quantitative
P4	Engineering^{+}	Case Study	Explo. & Desc.	Quantitative
P5	Empirical^{+}	Case Study	Descriptive	Qualitative
P6	Empirical	Case Study	Descriptive	Quantitative
P7	Empirical	Survey	Explo. & Desc.	Quan. & Qual.

*Non-structured survey (see Section 3.3.2)

$^{+}$Partially characterizable as *design science* (see Sections 3.3.4 and 3.3.5)

Finally, *the action research* is similar to a case study as it examines a contemporary phenomena in a particular context, but there is even less control as no distinct pre- and post-event analysis points exist. Rather, the approach's aim is to "influence or change some aspect of whatever is the focus of the research" (Robson, 2002). In concretion, there might be an initial setup comparable to a case study, but an 'observe–reflect–act–evaluate–modify–move in other directions' cycle is continuously applied to it, ever modifying the setup as new knowledge is produced Whitehead and McNiff (2006). In practice, action research often means developing and examining a solution dynamically as part of the problem context. Due to its more organic nature, distinction and examination of singular enhancement is, arguably, more troublesome.

Research characteristics also vary between research approaches. First, *the research objective* can be *exploratory, descriptive, explanatory,* or *improvement. Exploratory* research is concerned with scoping a particular research domain to identify and understand all variables affecting it. *Descriptive* research systematically establishes characteristics and functions for the research target. *Explanatory* research (or causal research) provides justification for a the existence of particular relationship between research target's variables. The control component of both the case study and the experiment allows exploratory objectives to be researched while the uncontrolled survey is used for more descriptive and explanatory objectives. The action research approach differs from these and meets only direct *improvement* objectives.

The research data gathered by the different approaches can be quantitative or qualitative. Data is quantitative if all of its values are known prior to measurement. For such data, statistical analysis can be performed in order to establish if there exists relationships. Thus, the experiment research

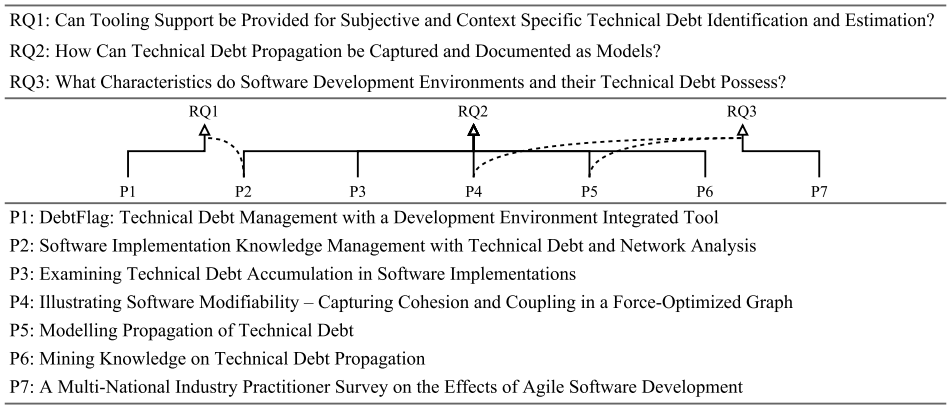


Figure 3.2: The thesis Publications *P1* through *P7* mapped to the Research Questions *RQ1* through *RQ3* they answer

approach deals only with quantitative data. Qualitative data are previously unknown and requires interpretation to be useful. As the "value" is dependent on the interpretation, no statistical analysis can be performed. Many, real world observations can only be recorded as qualitative data, and all research approaches, excluding the experiment, work with it. Quantitative and qualitative research data are also referred to as the fixed and flexible research designs respectively as often qualitative research data gathering encompasses some quantitative data as well. (Wohlin et al., 2012; Runeson and Höst, 2009)

3.3 Approach

The following will describe the research approach taken to overcome the set research questions. For each publication included into this thesis (*P1–P7*) a short *summary* is provided describing the essential contents of the paper. This is followed by describing the taken research *method* for each publication; in accordance with the general research methodology review provided in Section 3.2 (see Table 3.1). Finally, the publication under examination, is mapped against the set research questions (*RQ1–RQ3*) to highlight and discuss its *delivery*.

3.3.1 *P1*: DebtFlag: Technical Debt Management with a Development Environment Integrated Tool

Summary The first included Publication *P1* (Holvitie and Leppänen, 2013) is titled *DebtFlag: Technical Debt Management with a Development Environment Integrated Tool*. The publication introduces the DebtFlag tool

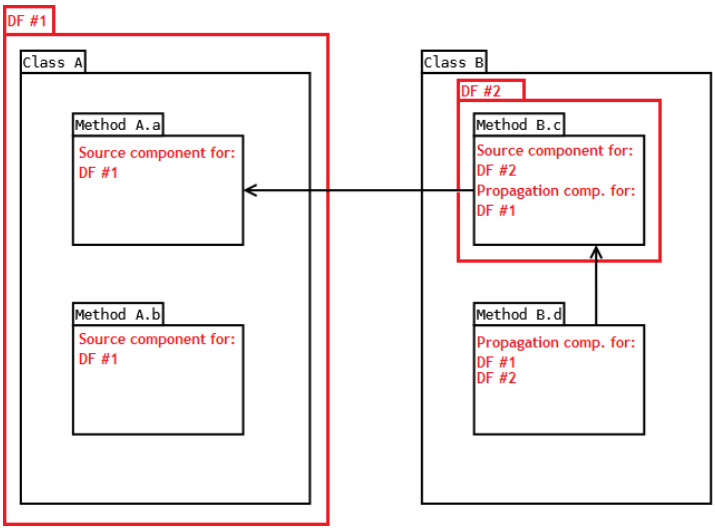


Figure 3.3: The DebtFlag mechanism (see Publication *P1*) © 2013 IEEE

which integrates into existing IDEs (Integrated Development Environment) in order to assist software development practitioners in capturing and managing technical debt. First half of the publication is theoretical, and it builds the DebtFlag mechanism (see Figure 3.3). The mechanism can be considered a mediator between existing research on technical debt management and the implementation practicalities. As basis for the mechanism, the publication reviews existing work on the area, identifies the Technical Debt Management Framework (TDMF; see Section 2.4), extends it in order to make it compatible with the hierarchical structures handled by IDEs (for example the package–class–method–sentence structures in common Abstract Syntax Trees (AST) like the ones used by the Java language[2].

The second half of the article encompasses the implementation of the DebtFlag mechanism in order to produce a tool to support it. The tool depicted in Figure 3.4 is created by extending the popular Eclipse IDE[3] with a plug-in that houses the implementation. The plug-in integrates the mechanism, which extends the TDMF, to the Eclipse JDT (Java Development Tooling) plug-ins. This is an example of attaching the mechanism to a provider of known semantics. Through this, the DebtFlag is capable of exploiting the semantics in order to provide automatic information production and maintenance. In addition to the plug-in, a web interface is also built which serves as a documentation storage and presentation platform for the data that is gathered by the individual plug-ins (one plug-in can be considered an interaction device used by one software development practitioner).

[2] `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`
[3] `https://eclipse.org/ide/`

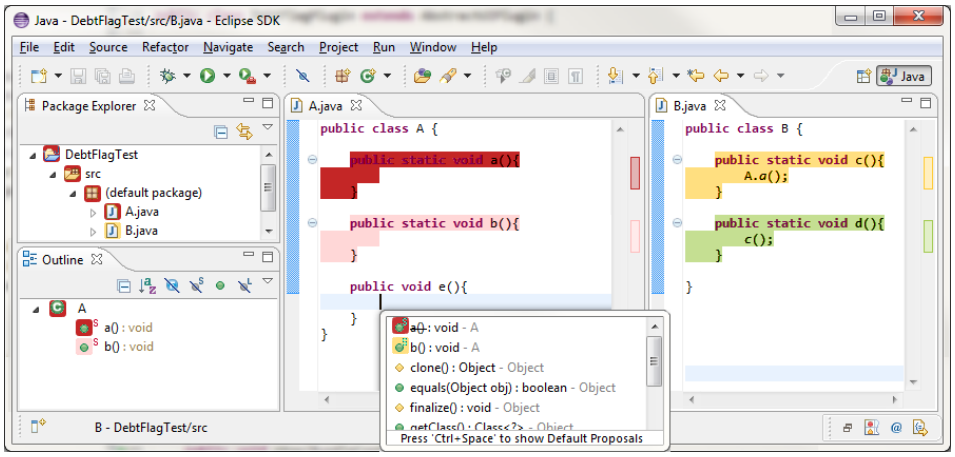


Figure 3.4: View of the DebtFlag tool implented as a plug-in for the Eclipse IDE (see Publication *P1*) © 2013 IEEE

Method In Publication *P1*, the DebtFlag mechanism and tool are produced via following the initial stages of the scientific method: existing literature is reviewed to conceive the improvements and they are followed by further revisions to justify the improvement. Validation on part of the complete tool is not provided in the publication as building the mechanism and the tool require several sub-components of the work to be fully defined, trialed, and validated prior to the complete tool being exposed for empirical validation. publications following this one (*P2* through *P7*) take this into account and each sub-component will be noted in their respective introduction. Regarding Publication *P1*, certain characteristics of the action research approach can be identified from the article, as the theoretical revisions of software development processes are conducted in order to pinpoint how the mechanisms, and tool implementing it, could enhance them. As such, for the Publication *P1*, the research objective is exploratory as new possibilities of improvement are discovered via interaction with the tool, and as the tool is used by practitioners as part of the discussed processes, the expected research data is qualitative by nature.

Delivery Publication *P1* meets Research Question *RQ1* which asks *Can Tooling Support be Provided for Subjective and Context Specific Technical Debt Identification and Estimation?* DebtFlag's direct integration into the Eclipse IDE exposes the tool's capturing processes directly to the software practitioner and hence makes it possible for the developer to document his or her subjective opinions about technical debt; his or her identification and estimation of particular technical debt instance. Further, as the DebtFlag functions as part of the IDE and communicates with the integrated devel-

opment tools (i.e., compilers, AST parsers, and word processors forming the Eclipse JDT) the operations committed by the practitioner can be supported at the context specific level. In practice this means that, firstly, the information produced by the developer can be captured whilst retaining the structure specified by the structure (i.e., identifying a Java class as a technical debt instance implicitly identifies all the class' members, like methods and variables, to be part of the instance as well). Second, production of the subjective information can be supported as the previously mentioned structures and their components can be offered to the tool's user when they form their description.

3.3.2 *P2*: Software Implementation Knowledge Management with Technical Debt and Network Analysis

Summary Publication *P2* (Holvitie, 2014), titled *Software Implementation Knowledge Management with Technical Debt and Network Analysis*, provides the theoretical grounds for Research Question *RQ2* in hypothesizing that "software implementation technical debt can be captured and maintained". The publication examines related work in the field in order to establish premises for the stated hypothesis and lays out three concrete objectives to overcome it. Linking the dissertation's Research Questions *RQ1* and *RQ2* together, the publication's objectives are to build 1) a technical debt management tool, 2) a static update model for it, and 3) a maintenance plan for the captured information.

Method In deriving its three objectives, Publication *P2* overgoes related research in the field. This survey acknowledges adjacent domains from which approaches can be combined in order to resolve challenges identified for the technical debt domain. Namely, use of tooling to be ready-to-document when notions about technical debt are made, network analysis, and programming theory to facilitate automated updating for the captured notions based on observed software evolution steps. The work committed in *P2* partially follows the engineering research method, as it observes existing solutions and introduces enhancements for them as combinations of reviewed approaches inclusive of the research groups previous contributions. The approach of the publication could hence be described to be a non-structured survey. The previously stated research objective is exploratory and the data consists from the author's subjective notions about reviewed research and is hence qualitative.

Delivery The technical debt management tool corresponds to the DebtFlag mechanism and tool introduced in Publication *P1*. The static update model is constructed as the end result of research presented in, upcoming,

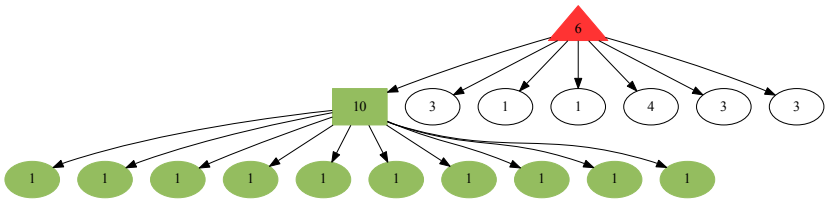


Figure 3.5: A Technical Debt Propagation Tree (TDPT) captured from the historical data of a software system implemented in Java (see Publication *P3*) © 2015 SERSC

Publications *P2* through *P5* from which the three first publications underlay the foundations for capturing technical debt and studying its structural characteristics in order to derive the static, technical debt propagation, update models. Publication *P5* captures the resulting modelling approach. Publication *P2*'s last objective describes a maintenance plan, wherein the derived models are integrated to the DebtFlag tool and it is operationalized to execute the maintenance plan in pre-existing software projects. Upcoming Publication *P7*, in fulfilling the dissertation's Research Question *RQ3*, characterizes the projects readying them for integration of the maintenance procedure while the evaluation of the process is described as the dissertation's future work.

3.3.3 *P3*: Examining Technical Debt Accumulation in Software Implementations

Summary Publication *P3* (Holvitie and Leppänen, 2015) is a multiple case study research into technical debt aggregation and it is titled *Examining Technical Debt Accumulation in Software Implementations*. The article operationalizes a method for extracting propagations of technical debt from historical data. The propagations describe how a particular technical debt instance, or part of an instance, has accrued in the software assets over a period of time; documented by the manipulated historical data. The historical data is comprised from two sets, the first set is a medium sized Java system used for educational purposes. The system has a diverse and well-documented development history which can be considered ideal for most retrospective software studies; the analysis of technical debt in this case. The second set comprises a selection of bugs and their accompanying historical data from a very large software development environment system developed mainly in Java.

The publication's main results are comprised from identifying and characterizing the different propagation capabilities observed for the followed technical debt instances; recorded as mutual characteristics of captured Technical Debt Propagation Trees (TDPT). The TDPT, of which an example can be seen in Figure 3.5, is a tree graph which is also referred to as the technical debt propagation path. Further, they are compared against standing discoveries and hypotheses identified from related work. There are two motives for conducting the research in Publication *P3*. Firstly, the discovered propagation capabilities extend the knowledge in this research field; aiming for better understanding of technical debt's abilities and effects on software processes. Second, observing that the two independent sets highlight similar propagation capabilities provides further justification and an initial starting point for constructing technical debt propagation models.

Method The research conducted by Publication *P3 Examining Technical Debt Accumulation in Software Implementations* follows the scientific method. Revision of existing literature on technical debt propagation cases and tracking attempts builds a model or a theory of a technical debt propagation system. Empirical observations in the publication not only provide justification for the system, but also enhance it by demonstrating more in-depth details for the propagation. The details inhere to the existing propagation knowledge, but reveal new dimensions for consideration when assessing technical debt propagation. The used research approach is case study, as two independent, contemporary phenomena are studied in their respective contexts. While the resulting data is quantitative (i.e., propagation graphs and integer values marking dependency counts) parts of them are produced through means of subjective assessment and as such can be considered qualitative. This makes the research design flexible for the study in question. The research objective is descriptive as the intent of the two case studies is to characterize technical debt propagation as exhaustively as possible for the considered context.

Delivery Capturing technical debt propagation in models is pursued by Research Question *RQ2*, and, as such, the publication in question can be seen to pave the way towards meeting it. Using Publication *P3*, amongst other, as a starting point, upcoming Publication *P5* builds the technical debt models in order to meet *RQ2*. Further, the models are one of the subcomponents identified for the DebtFlag mechanism in Publication *P1* and they are used by the DebtFlag tool to provide automatic maintenance for the manually captured subjective technical debt information.

3.3.4 *P4*: Illustrating Software Modifiability – Capturing Cohesion and Coupling in a Force-Optimized Graph

Summary The previous Publication *P3* examined technical debt propagation for a quantity of software assets that remained manually approachable. However, in order to exhaustively understand the capabilities of technical debt, the whole system should be subjected for similar analysis. In these scenarios, the number of considered assets quickly grows to several thousands; making manual inspection infeasible. Hence, Publication *P4* (Holvitie and Leppänen, 2014) introduces an automatic and scalable approach in *Illustrating Software Modifiability – Capturing Cohesion and Coupling in a Force-Optimized Graph*. Here, software assets and relations between them are modelled as a directed graph. The edges of the graph record the particular relation strength as a "force" (e.g., number of reference invocations) between two particular vertices (e.g., program methods). As a default feature of graphs, the inspection granularity can also be changed prior to running the chosen layout algorithm: moving from method to class level merely models the class as one vertex which inherits all relations from the class-forming methods.

Having finalized the graph, a force-directed layout algorithm is executed on it. The publication overgoes different algorithmic options prior to deciding on the ForceAtlas2[4]. After the algorithm converges, the graph is laid out in a force-optimized manner. This means that a globally acceptable (not necessarily globally optimal) energy state exists in the graph. In other words, there are very few to no vertices that could be repositioned in the graph resulting in the graph to display smaller total tension between all of its vertices. In this way, the publication is able to identify with a graph—consisting from a few to hundreds of thousands of vertices representing software assets—structures and clusters that are meaningful from the point-of-view of the entire modeled system. The method outcome is demonstrated for an extreme case (1.50M directed edges) in Figure 3.6 where the Eclipse IDE implementation is layed out.

Method The research method used in Publication *P4* is the engineering method. Previous literature is reviewed in order to establish existing solutions from software change analysis, technical debt propagation modelling, and graph theory. An enhanced method is then proposed on top of them in order to exploit the previously established methods in a new context. Initial validation is provided in a case study, wherein quantitative analysis is provided by overgoing the structure and cluster highlighting features

[4] `http://www.medialab.sciences-po.fr/publications/Jacomy_Heymann_Venturini-Force_Atlas2.pdf` (Draft version, Jacomy et al., 2012)

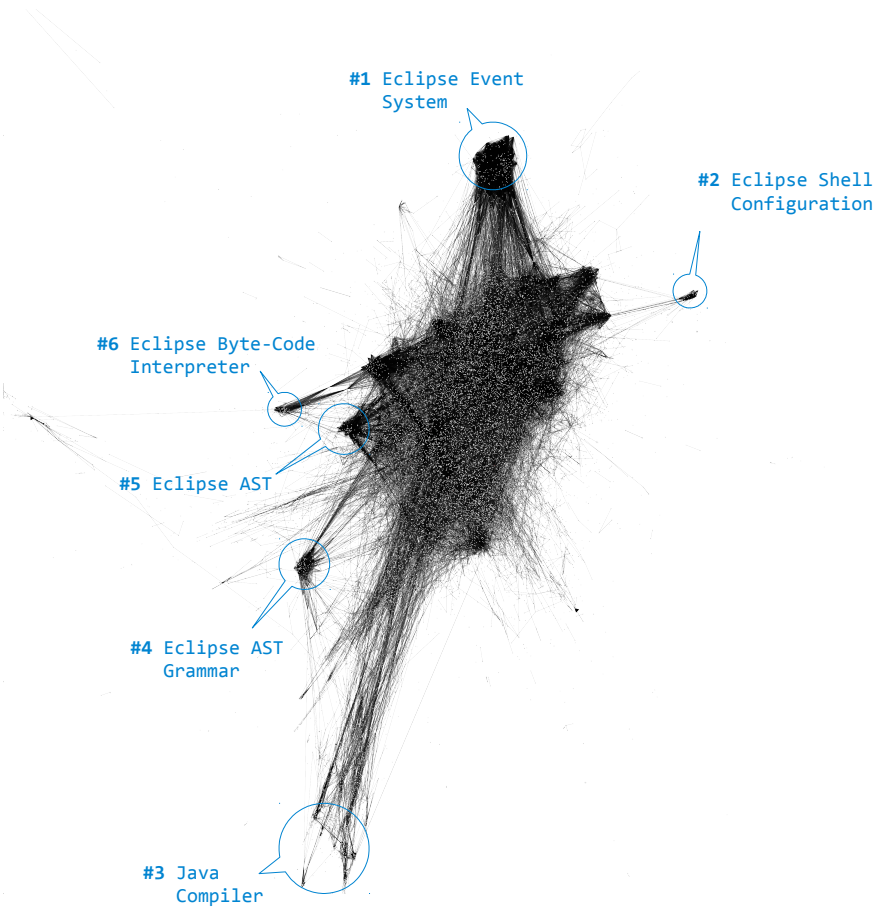


Figure 3.6: The Eclipse dependency tree displayed in a force-optimized graph with distinct clusters highlighted (see Publication *P4*) © 2014 IEEE

of the graph and comparing them against the documented architecture of the system modelled in the graph. Aforedescribed actions partially adhere to the design science research methodology (DSRM) proposed by Peffers et al. (2007). Especially, the four first activities of DSRM (i.e., *problem identification and motivation, define objectives for a solution, design and development,* and *demonstration*) are present. Based on this, the research could be described more as design science, but meticulous adherence to the DSRM process and robust evaluation are not provided as part of the publication. To that end, the research objective is thus both exploratory and descriptive in nature as no graph approach has previously been applied for the technical debt context and comparisons to well established cohesion and coupling measures are provided to establish the approach against similar research.

Delivery The structures and clusters identified from the force-optimized graph further carry the technical debt propagation research onwards and closer to the modelling Research Question *RQ2*. First, as the graph is built from the same assets from which Publication *P3* was able to observe singular technical debt propagation characteristics, this method is seen to provide a way to extend observation of the characteristics to larger structures. Second, as the force-optimization highlights structures which are important—relative to the surrounding structures—the method provides a way to link and examine the role and effect that these structures have on the previously established propagation characteristics. Third, as part of the graph forming capabilities, Publication *P4* demonstrates highlighting features in order to distinguish groups of interest from the whole graph—and thus from the entire modeled system. This allows contrasting the manually produced TDPTs against the entire system structure. This can be used to further argue for or against researched propagatory features.

3.3.5 *P5*: Modelling Propagation of Technical Debt

Summary Publication *P5* (Holvitie, Licorish and Leppänen, 2016b) builds on the technical debt propagation observation methods established in the previous publications. *P5* defines a process for forming technical debt models from empirical data. In this, technical debt propagations are considered to realize as paths that are formed from technical debt channels (see Figure 3.7). The technical debt channels are defined as mediums that have a source and a destination point that interchange technical debt information. The article requires that the historical data, from which the technical channels are built, provides such time and partition granularity that the evolution of a single software asset can be observed as a sequence of states. This is a requirement of the modelling process to exhaustively and unambiguously

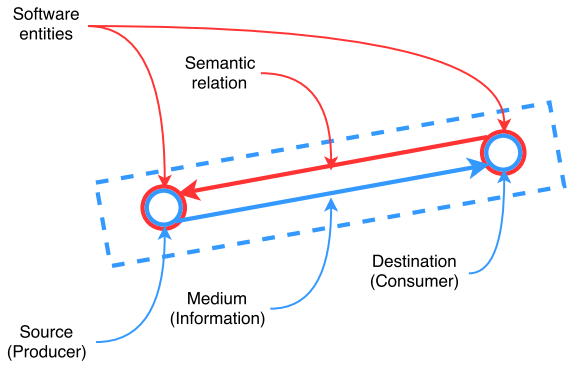


Figure 3.7: Demonstration of two interconnected software assets overlaid with the components of a technical debt channel (see Publication *P5*) © 2015 IEEE

capture all technical debt channels. Further, the source and destination points are dictated as cause-and-effect relations for the finding of which the publication presents formulae.

Groups of technical debt channels are formed based on their technical debt propagation capabilities which are captured by the channels' sub-components. As the sub-components still manifest some context bound properties which do not affect their propagation capabilities, the last stage of the modelling approach abstracts this information off from the group of technical debt channels. The resulting technical debt propagation model is the abstracted channel description with a source, a destination, and an information interchange event that implies technical debt inclined change (example provided in Table 3.2). Definition of the change is based on Schmid (2013) and Kagdi et al. (2007) previous work on heightened—with respect to an optimal system state—resource consumption to implement a change.

Method Publication *P5* follows the empirical method in conducting research. An initial model, the technical debt channel description, is conceived based on reviewed, prior knowledge. A process is defined, the technical debt model forming process, to accumulate empirical evidence for validating the model. Thus far, the publication's research process follows the DSRM (see previous Section 3.3.4). However, the publication then proceeds to provide initial validation for the model by applying the process on top of the descriptive research committed in Publication *P3* which leads the publication away from committing the later activities of the DSRM. By doing this, the publication is, however, able to successfully present results. The research approach used to gather the empirical results is the case study. The empirical data source is the same as for Publication *P3*: a bug tracker and version

Table 3.2: A Technical Debt Model constructed from technical debt channels recorded for software assets implemented in the Java language (see Publication *P5*) © 2015 IEEE

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

control system for an open source software project. The case study limits to one particular bug instance as Publication *P5* demonstrates the modelling approach in practice by applying it on to this instance. Ideally, the research objective would be explanatory as the model pursues an exhaustive answer to the question of "how much technical debt is accumulated in a particular section of the technical debt propagation path?". The publication, however, remains at a descriptive level as in-depth empirical validation is required to exclude remaining threads to validity; especially in relation to the subjective evaluation executed for the cause-and-effect relations. Noting the previous, it can also be determined that the research data is qualitative in nature.

Delivery Publication *P5* titled *Modelling Propagation of Technical Debt* answers *RQ2* that prompts: *How Can Technical Debt Propagation be Captured and Documented as Models?* The publication describes—the previously described—three step processes for identifying technical debt propagation from historical data, translating the propagation into channel descriptions, and comparing the channels' propagatory properties to form classes that can be documented as models via context reduction.

3.3.6 *P6*: Mining Knowledge on Technical Debt Propagation

Summary Publication *P6* (Suovuo, Holvitie, Smed and Leppänen, 2015) is titled *Mining Knowledge on Technical Debt Propagation* and it can be seen to extend the characterization of efforts of the modelling Publication *P5*. Publication *P6* applies the Mining Software Repositories (MSR) approach to capture technical debt propagation and effort as a function of time. This longitudinal study of technical debt's effect on projects over several years is possible through the method introduced by the publication. Herein, the MSR is directed at open-source software repositories and it is allowed to crawl through years of historical data (including repository commits, commit

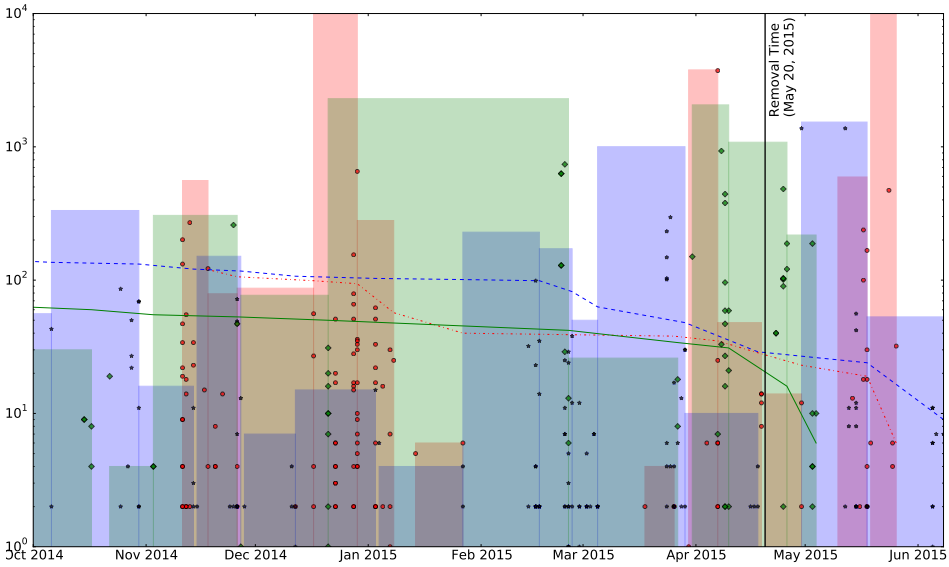


Figure 3.8: Three open-source projects demonstrating the impact invoked by a change in an external API (see Publication *P6*)

messages, Git[5] and GitHub[6] specific data such as fork, blame, discussions, and project voting data).

The historical data is then compared against clearly external events to record their impact in the historical data. The event data records major version changes in the APIs (Application Programming Interface) used by the crawled open-source projects. These changes imply potential technical debt accumulation to the projects, as they must adapt the project to the impending changes. Example of an external-API-invoked technical debt accumulation is provided in Figure 3.8. Gathering enough data points allows us to exclude noise (e.g., fluctuations in the development velocity of individual projects) and to pinpoint the tracked effects. Further, if an effect is recorded for consecutive changes in the same API, it is possible to observe technical debt "learning" for the projects as they can be assumed to expect these API changes after having faced a few.

Pinpointing the technical debt inducer, the change in the external API, allows also the identification of related software contexts. For example, the publication demonstrates that a change made in a project in response to an API changing has affected assets created in JavaScript[7] and as JPEG[8]. This provides evidence for argumenting for a relationship to exist between these

[5] `https://git-scm.com/`
[6] `https://github.com/`
[7] `https://developer.mozilla.org/en-US/docs/Web/JavaScript`
[8] `https://jpeg.org/jpeg/`

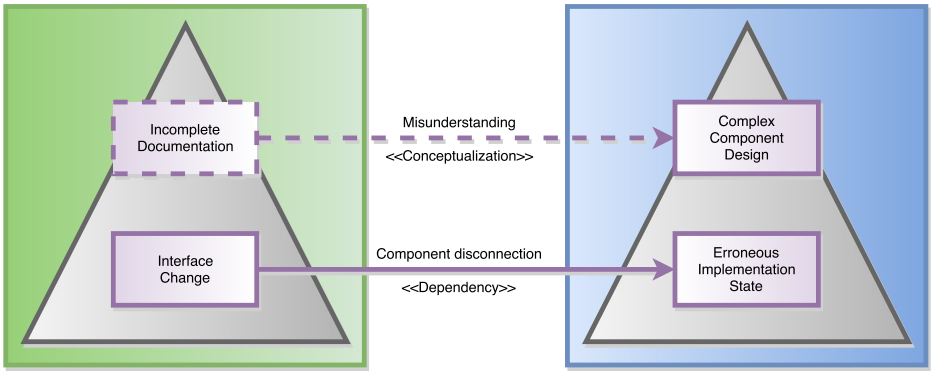


Figure 3.9: Demonstration of the explicit and implicit channel existing between two software assets (see Publication *P6*)

two contexts. This is an important starting point for an in-depth analysis into verifying and revealing the intrinsics of the relationship.

Publication *P6* also discusses further in-depth issues relating to the technical debt modelling. Firstly, the division of implicit and explicit channels is introduced (see Figure 3.9). An explicit technical debt channel is one for which the software assets' technology contexts provide semantics for. Thus, the channel will explicitly propagate technical debt with the described mechanism (e.g., an inheritance relationship between two program classes). The implicit channel is an identical relation between software assets, but no pre-existing semantics can be used to describe the information transfer here. An example of this is the relation where a design asset describes an implementation asset. The ability to conceptualize is used by the software developer to translate the design asset to an implementation asset. This is an implicit channel as no pre-existing semantics can describe this process. The publication discusses this as a challenge for technical debt modelling.

Method & Delivery The publication in question further contributes towards Research Question *RQ2*. The research is conducted using the empirical method as the publication firstly describes the modelling expectations, the process suggested by the publication is then used to gather empirical evidence to produce data for its validation. Initial validation is provided in the paper by demonstrating the effects of an API change on three analyzed open-source projects. The research approach used to gather the data is the case study, and the research objective is descriptive in explaining further characteristics for the targeted technical debt propagation trending and "learning" effects. The data is quantitative in nature and hence allows experimental validation to extend this research with causality analysis.

3.3.7 *P7*: Technical Debt and Agile Software Development Practices and Processes: An Industry Practitioner Survey

Summary Publication *P7* (Holvitie, Licorish, Spinola, Hyrynsalmi, Buchan, Mendes, MacDonell and Leppänen, 2017) titled *Technical Debt and Agile Software Development Practices and Processes: An Industry Practitioner Survey* conducts a multi-national software practitioner survey. The survey was designed to reflect both, software development environment and technical debt instance characterization needs identified to be lacking from the research domain. In order to characterize software development environments, the survey design captured details regarding the respondent as well as his or her project, team, and organization. Background details queried ranged from years of experience to roles assumed in projects as well as team size. For software development, the survey queried which methods were applied in the respondent's project—the one he/she was most associated with. Going into more detail, it was established which software development practices and processes, possibly from the adopted methods, were being applied and to what degree. The survey then queried the respondents about their prior knowledge and definition of technical debt. Afterwards, the respondents were acquainted with current definitions and explanations regarding technical debt and its effects in order to harmonize the respondents' answers for the latter part of the survey.

The latter part of the survey focused on technical debt. First, the survey queried the respondents how they perceived the adopted software development methods, practices, and process to affect technical debt management (see Figure 3.10). Second, the respondents were given a chance to describe an actual instance of technical debt that was affecting their work. The queried description for the instance captured the origins for the instance's emergence (e.g., legacy software assets), and the reasons for the instance's existence (i.e., the causes stated in Kruchten et al. (2012) *Technical Debt Landscape* like "Inadequate architecture" or "Missing documentation" (see adapted selection set and indicated frequencies depicted in Figure 3.11). It can also be noted that, among the captured reasons in Figure 3.11, there are notions ("New features are required" and "Additional functionality is required") which are generally not regarded to contribute technical debt in software projects Kruchten et al. (2012). Additional analysis is required to understand why some stakeholders still perceive this to be the case. Further, it was established how the technical debt instance manifested in the software development. This included capturing the parts of the software development life-cycle (see Section 2.2) that were being affected by technical debt, the dynamics of the instance when development in these areas was continued, and reflecting the dynamics back to the effects felt for the instance.

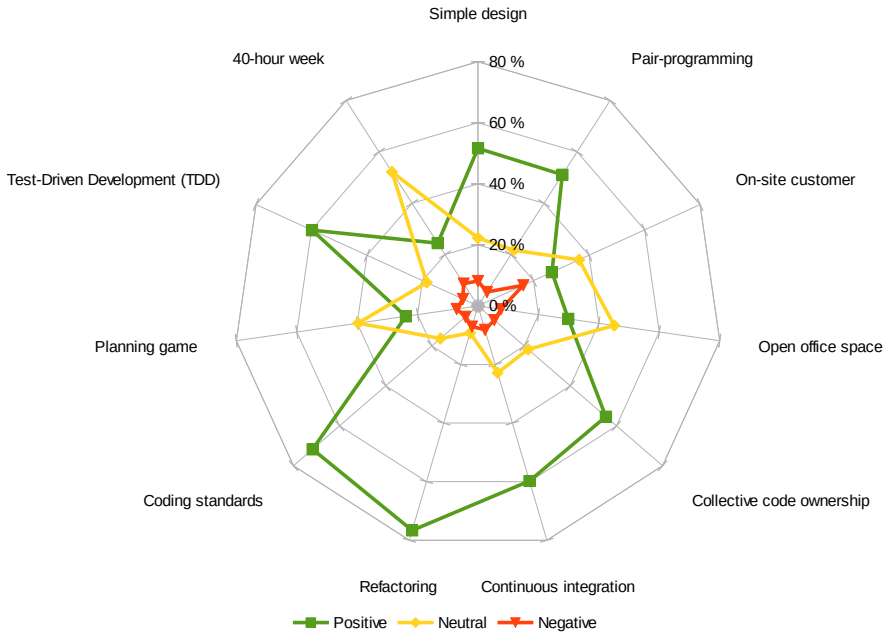


Figure 3.10: Spider graph documenting the perceived effect of different agile practices on the management of technical debt (see Publication *P7*)

Method Publication *P7* uses the empirical research method to answer research questions that pose a clear model of technical debt's association into software development. The research approach used is the survey. The data collected by it is both quantitative and qualitative, but the survey's questionnaire design ensures that research questions can be answered via manipulation of quantitative data only. This makes it possible to have the research objective both descriptive and explanatory. Descriptive parts of the survey pave way for integration of previous contributions; like the DebtFlag tool and application of the force-optimized graph. The explanatory part identifies the required characteristics for software development methods and certain characteristics of technical debt via statistic analysis. Causality implied by statistically significant results establishes a base for future research invoked in this dissertation (e.g., the integration of contributions from the previously discussed publications into practice).

Delivery Research Question *RQ3* asks *What Characteristics do Software Development Environments and their Technical Debt Possess?* Publication *P7* answers this question with the multi-national software practitioner survey. The aforedescribed design of the survey first addressed the software development environments, their individuals, and the individuals' backgrounds

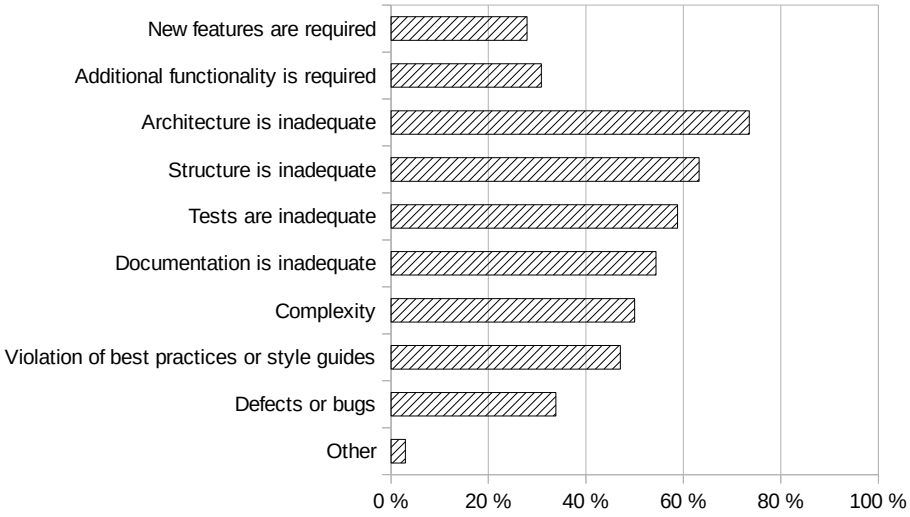


Figure 3.11: Causes for the existence of technical debt in software projects recorded as a set selection from the "Technical Debt Landscape" (Kruchten et al., 2012) (see Publication *P7*)

in order to exhaustively map the environment in which technical debt management works and into which further management solutions are expected to integrate to. The second part of the survey captures intrinsic details about technical debt instances that exist and affect real software projects. This information is vital for both the management enhancement efforts, but also records novel empirical data for the technical debt research domain.

Chapter 4

Conclusions

Chapter 2 built the theoretical background for the work presented in this dissertation. The technical debt concept was defined from multiple views points in order to exhaustively characterize it. To provide a working context, this was followed by reviewing core components and processes related to software development. Inclusive to this were the software development life-cycle, the software assets produced in it, as well as the procedures and roles assumed by practitioners implementing the life-cycle. Contrasting the technical debt definition to the one provided for software development allowed the dissertation to address how technical debt emerges and affects all software undertakings, regardless of their execution. This provided a backdrop against which technical debt management could be introduced. As previous matters were discussed through a revision of current research, a number of challenges, future work, and implications derivable from the combined research could be seen to emerge. These were discussed as motivations (*M1* through *M5*) for additional research pursuing efficient technical debt management.

Chapter 3 described the research undertaken for the dissertation's topic of *Technical Debt in Software Development – Examining Premises and Overcoming Implementation for Efficient Management*. As the presented Motivations *M1* through *M5* could be seen to hold tremendous effect on the implementation of technical debt management for software development projects, based on them, the chapter defined three Research Questions *RQ1* through *RQ3* aimed at *Examining Premises and Overcoming Implementation for Efficient Management* of technical debt. The motivations influenced the design of individual research questions: Research Question *RQ1* (*Can Tooling Support be Provided for Subjective and Context Specific Technical Debt Identification and Estimation?*) addressed Motivation *M1*, *RQ2* (*How Can Technical Debt Propagation be Captured and Documented as Models?*) focused on *M2* and *M3*, and *RQ3* (*What Characteristics do Software Devel-*

opment Environments and their Technical Debt Possess?) acknowledged *M4* and *M5*.

The approach taken to overcome the stated research questions constitutes the seven, *P1* through *P7*, individual publications included into this dissertation. Publication *P1* was capable of overcoming Research Question *RQ1*, while a total of four publications, *P2* through *P5*, lead to Publication *P6* overcoming Research Question *RQ2*. The bipartite Research Question *RQ3* is addressed by the final seventh Publication *P7*. The section following this continues the conclusions for this dissertation by listing the contributions that aforementioned publications have produced; inclusive of supplementary theoretical and practical contributions derived parallel to solutions for the original research questions. The final section of the chapter in question selectively revisits the work describe in this thesis to establish items of future work. A mapping between the discussed Contributions, research questions, and included publications is captured in Table 4.1.

4.1 Contributions

The following revises the contributions this dissertation has made towards *Examining Premises and Overcoming Implementation for Efficient Management* within the context of *Technical Debt in Software Development*. A total of fifteen contributions (*C1–C15*) are described here in three parts. First part, in Section 4.1.1, is the most significant one as it discusses the contributions (*C1–C7*) that contribute answers to the three research questions (*RQ1–RQ3*) set forth in this dissertation. There also supplementary results which are produced in the included publications when meeting the research questions. Second part, beginning of Section 4.1.2, discusses those supplementary Contributions (*C8–C11*) with theoretical value while the third part, end of Section 4.1.2, discusses those (*C12–C15*) with practical value. Theoretical aspects advance the research field whilst the practical contributions offer executable solutions for practitioners. Finally, whilst not concrete contributions per se, the author would like to acknowledge the research-additive-role of the presented motivations (*M1–M5*): contributions revised below are unable to exhaustively encompass the challenges highlighted by the motivations, and, as such, they can be seen to contribute future work targets for the technical debt research field.

4.1.1 Research Questions

The dissertation set forth three research questions in Section 3.1 and presented an approach for overcoming them in Section 3.3 via the included publications. The following reviews the contributed solutions for the three research questions.

Table 4.1: Fifteen individual Contributions (*C1–C15*) are identified from the discussed publications (*P1–P7*). Major contributions of the publications pursue answering the posed research questions (*RQ1–RQ3*). Remaining contributions are supplementary and carry either theoretical or practical value

Relation	**Contribution**	**Discussion**	**In Publ.**
RQ1	*C1*: DebtFlag Mechanism	p. 70	*P1*
	C2: DebtFlag Tool	p. 71	*P1*
RQ2	*C3*: Technical Debt Characterization	p. 72	*P2–P4*, *P6*
	C4: Technical Debt Channel Description	p. 72	*P5*
	C5: Technical Debt Propagation Identification Procedure	p. 72	*P3*, *P5*
	C6: Technical Debt Propagation Model Formation Procedure	p. 73	*P3*, *P5*
RQ3	*C7*: Software Development Environment and their Technical Debt Characterization	p. 73	*P7*
Supplem., Theoretical	*C8*: Extension of the Technical Debt Documentation Structure	p. 75	*P1*
	C9: Technical Debt Propagation Characterization	p. 77	*P3*
	C10: Technical Debt Propagation Channel Described as an Information Medium	p. 77	*P5*
	C11: Empirical Verification of Technical Debt Properties Practical Aspects	p. 78	*P7*
Supplem., Practical	*C12*: Context Independent Technical Debt Management	p. 79	*P1*
	C13: Force-Optimal Layout of AST Information	p. 80	*P4*
	C14: Technical Debt Propagation Model Library	p. 81	*P5*
	C15: In-depth Characterization of Software Organizations	p. 82	*P7*

Research Question *RQ1*

The first research question, *Can Tooling Support be Provided for Subjective and Context Specific Technical Debt Identification and Estimation?*, was successfully answered by Publication *P1* as it defined the DebtFlag mechanism (*C1*) and the DebtFlag tool (*C2*). The mechanism establishes the prerequisites for tooling support, whilst the tool itself provides an example of the mechanism's implementation. The end result is TDMF compliant, it supports manual technical debt identification for maximal coverage, and enables efficient maintenance for the captured information via automatically assessable propagation models. Empirical validation is, however, required for the solution proposed in *P1*.

C1: The DebtFlag Mechanism contributes a technical debt documentation structure, a requirements set for automated information maintenance, and two management procedures. The documentation structure describes how a technical debt item (TDI; see Table 2.2) can be captured from pre-existing and pre-specified software assets: a technical debt item is considered a cluster of technical debt elements that adhere to the contexts' element definitions. Adherence to the pre-existing context semantics allows retaining the context knowledge. This is also the first requirement for automatic information maintenance. As technical debt is accumulated through relationships between software assets, the automation requires that the mechanism is able to identify the atomic software assets forming the relationships. Hence there needs to exist translations that describe for higher level assets the lowest level, atomic assets capable of realizing the technical debt accumulation in them (i.e., technical debt items are formed from clusters of elements as noted in *P1*). The second automation requirement is the ability to apply the software assets' context semantics for the captured technical debt. Application of the semantics is enabled as the debt is captured clusters (i.e., low level software assets) defined of elements defined by the context. However, application of the semantics would only model the semantics' functionality. Hence, the semantics are associated with rule sets that modify the existing semantics to describe how they realize technical debt accumulation. These rule sets correspond to the technical debt models discussed in Research Question *RQ2*.

Two management procedures were defined by the DebtFlag mechanism. The first is a project and organization level management procedure which is based on the documentation structure supporting the TDMF through maintaining the TDL (Technical Debt List; see Section 2.4). This is a representation of the project's identified technical debt. The procedure allows application of further, TDMF compliant, management procedures, or for direct use of the TDL as a technical debt knowledge source for decision

making. The second management procedure is micro-management. As the documentation structure records TDIs as clusters of software elements, it is possible to provide a presentation for the TDIs directly at the assets' development-time. Various methods, like restriction and visualization, can be used in the integrated development environment to indicate that the asset is either a root or a propagation element of a technical debt item. This informs practitioners working with associated software assets about the matter and is expected to lead to technical debt management with minimum added effort.

C2: The DebtFlag Tool in realizing the DebtFlag mechanism, the tool contributes a viable path for implementing the mechanisms' previously described procedures. The tool's first part, the IDE plug-in, demonstrates how the described documentation structure is retained and enforced when working with program code software assets. Further, it shows how rule sets—technical debt models described in the next section—can be input for automating the technical debt information maintenance, and it builds a presentation from restriction and visualization for the TDIs to enable their micro-management. The second part of the tool is the web interface which shows how the information gathered from individual practitioners, the user of the DebtFlag plug-ins, can be accumulated to provide a project- or organization-wide view into the technical debt situation. Further, it enables technical debt decisions to be propagated back to the plug-ins as TDIs forming the TDL can be modified (e.g., their severity or estimates can be changed) from the web-interface; thus demonstrating a linkage between the high level management and micro-management.

Research Question *RQ2*

The second research question required capturing technical debt propagation in models, and, together with the groundwork in Publications *P2* through *P4* and with further characterization in *P6*, Publication *P5* contributed the procedure for this. The full procedure has several sub-components: a description of a technical debt channel, a procedure for associating technical debt inclined changes to identify propagation through the channels, and a procedure for abstracting the channels into technical debt models. Publications *P2* through *P6* successfully answer Research Question *RQ2*. They describe what contributes technical debt related information from a software history and the manner in which it can be found. An approach is derived where cause-and-effect relations are identified from the historical information in order to establish propagation for a technical debt instance. This is followed by a description of a technical debt channel with its features; finally enabling classifying similarly typed channels into technical debt models.

C3: Technical Debt Characterization approach established in *P2* and committed in *P3*, serve the ground work for examining how technical debt manifests in software. This is an important prerequisite for the upcoming technical debt channel description and modelling endeavours as it highlights the possible avenues of research for capturing technical debt. Namely, *P3* establishes the close relation between technical debt accumulation and dependency propagation. Further, it demonstrates the viability of sourcing version control repositories for possible technical debt instances, and, as such, sets the stages for further technical debt tracking procedures.

C4: Technical Debt Channel Description provides the necessary understanding required to associate software assets to actions and effects of technical debt. The description (see Figure 3.7 at page 60) states that a technical debt channel is formed from a source, a medium, and a destination. Both the source and destination component are software assets, and they both follow the atomic requirement as described by the previous section. In order to function as a technical debt channel, the description requires that the information interchanged by the components *indicates heightened resource consumption* (heightened as per Schmid (2013) definition of technical debt accumulation when a software asset consumes more resources in comparison to ideal when it evolves). This channel characteristic is an important pre-condition, as it implicitly states that technical debt propagates to a new software asset only when the propagation-inducing-asset is related to the new one, and when the propagation-inducing-asset manifests a change that is externally observable: if a change within a software asset is confined and not observable, it can not induce technical debt propagation.

C5: Technical Debt Propagation Identification Procedure is described, especially, in Publications *P3* and *P5*. It is a procedure which associates technical debt inclined changes together in order to produce propagation paths (i.e., linked technical debt channels). The procedure works on historical software change data. The data has to capture the change history for the software assets in addition to the reasoning for the changes (generally this is recorded by the version control system, their log entries, and possible other reporting system in use by the project or the organization). The subjective, reasoning data is reviewed to pinpoint technical debt motivated actions. The software change data is then queried for the first change invoked by these actions. The procedure then iteratively accumulates technical debt propagation paths for these "root causes" by associating further changes on cause-and-effect basis. The cause-and-effect relation corresponds to realization of the aforedescribed technical debt channel, and, as such, two changes form a cause-effect pair if the effect change can be seen to exist in

the software change history only because the cause change has invoked it. Importance of the contributed cause-and-effect associating approach should be highlighted, as it is the only approach capable of capturing all forms of technical debt propagation due to the use of subjective assessment.

C6: Technical Debt Propagation Model Formation Procedure is started by identifying a class of technical debt channels. This is achieved by reviewing their source, destination, and medium components to establish if they define similar propagation capabilities. The model is then formed through a simple abstraction process that reduces all project and organization context specific information—which does not affect propagation characteristics—away from the classes instances (i.e., the channels). As the models' source and destination components are software assets, the models are automatically assessible in scenarios where predefined semantics exist for the modelled component pair; the medium component providing further ruling indicating, for example, if the technical debt diminishes as part of transition between components.

The research question can be seen to invoke mainly research interest as it describes accumulation of models describing technical debt propagation. This is a highly cumbersome task, and, as such, difficult to justify to be done midst software development. However, singular executions of the procedure may still be done within a running project to further describe or document problematic or exotic cases of technical debt propagation for further review.

Research Question *RQ3*

The third Research Question *RQ3* asked *What Characteristics do Software Development Environments and their Technical Debt Possess?* in order to establish an interface for introducing further technical debt management enhancements. Publication *P7* answered this in contributing a detailed multinational survey characterizing software development practitioners and their environments position in relation to technical debt in addition to capturing detailed descriptions of technical debt instances currently affecting existing software projects and organizations.

C7: Software Development Environment and their Technical Debt Characterization is committed in Publication *P7*'s survey. The survey's description of individual practitioners can be seen to contribute valuable information to software organizations in the form of describing the knowledge level and adjustment towards technical debt in relation to individuals backgrounds. Notably, the survey demonstrated that practitioners were implicitly aware of technical debt, as they assume a low level of technical debt knowledge but adhered closely to presented definitions of the concept.

This could be seen as a reason for why many of the practitioners thought the concept to be highly usable in various scenarios but reported very scarce current usage. Further, common understanding of the technical debt concept is indicative of its usability in narrowing the communication gap. The communication gap describes a common nuisance of software development projects wherein the technically oriented staff (e.g., developers) and other organization members have a difficulty in effectively communicating matters due to their greatly differing working contexts.

Regarding software development environments, the survey contributes a number of insights relating to how different software development practices, processes, and process artifacts—that form the software development methods—interact with technical debt. The study found a number from the practices and processes to have an effect on technical debt. Especially, agile practices and processes which verified and maintained the structure and clarity of software assets had a considerably positive perceived effect on technical debt management. Interestingly, the communication gap was likely demonstrated for practices which involved multiple stakeholder groups, as these practices demonstrated the most diverse opinion spread. This part of the survey contributes existing software undertakings with knowledge regarding how the queried practices and processes affect technical debt management. This also exposed integration points for further research (i.e., retrospectives' and iteration backlogs' with perceived highly positive effect on technical debt).

Finally, the last part of the survey contributed a detailed description of technical debt instances currently existing and affecting software development undertakings. As described in Section 3.3.7, the origins, causes, residence, development dynamics, and effects were established for the instances. Notably, instances demonstrated variety on all previous counts which indicated that they were challenging, "translucent", management targets. Of high importance was the notion that a single technical debt instance was seen to have an effect on multiple phases from the software development lifecycle (the design and implementation phases combination being the most common one). It was also alarming to note that the instances generally grew in size as a result of continued development which also increased their negative effect on the project. However, from the point of view of management efforts, an important commonality was discovered in circa 75% of the instances having origins in legacy software assets. This notable similarity between the instances leads to the article suggesting that software developments should take especial care with legacy asset management whilst research should pursue narrowing the gap between the legacy and technical debt domains. The instance characterizations also provide a library for software undertakings to refer in estimating technical debt and future research to trial prominent management solutions on.

In summary, Publication *P7* successfully answers Research Question *RQ3*'s question *What Characteristics do Software Development Environments and their Technical Debt Possess?* The previous described how a multi-national dataset was gathered to establish background details, expertise, and adopted methods for a variety of software organizations through the individuals working in them. The survey carried on to identify the status of technical debt knowledge within the industry prior to gathering evidence on concrete technical debt instances with their causes, origins, and development dynamics and effects. This information provides an interface which describes the environment to which technical debt management approaches and tools must adapt to in addition to identifying and describing their targets, the technical debt instances.

4.1.2 Supplementary Outcomes

The following revises the results produced during this dissertation project that are supplementary in nature. The thesis has included seven publications that meet the dissertation's Research Questions *RQ1* through *RQ3*. In overcoming these, the publications contribute additional research outcomes that are of value in the associated domains. The following revision of these results is divided into theoretical and practical aspects.

Theoretical Aspects

Theoretical aspects are considered to be contributions that are not directly applicable in practice, but carry other value. Mainly, they add to the research committed in the respective domains. Following exclusively discusses those outcomes, from the publications included in the thesis, that are theoretical in nature.

C8: Extension of the Technical Debt Documentation Structure
Publication *P1*, *DebtFlag: Technical Debt Management with a Development Environment Integrated Tool*, introduces the DebtFlag mechanism in meeting *RQ1*. The mechanism extends the structural definition of a TDI (see Figure 4.1, and produces two theoretical supplementary results. First, allowing a TDI to be defined as a set of technical debt elements—introduced as DebtFlag elements in the publication—connects the TDMF into pre-existing software contexts. The eligible benefits from this, such as auto-updating and -propagating, were already discussed in Section 4.1.1. The additional value comes in the DebtFlag elements implicitly documenting and manifesting, through the mechanism, the technical debt's capability of merging and disbanding (discussed as mechanism *M5.b* in Section 2.5.5). As further technical debt items are defined, the software assets evolve and propagate the

```
Technical Debt List
> Technical Debt Item #1
    > DebtFlag #1
        - Time and Date
        - Author
        - Location
        - Description
        - Type Declaration
            > Technical Debt Type #1
                - Name
                - Description
                - Context
                - Propagation rules
            > Technical Debt Type 2
                ...
    > DebtFlag #2
        ...
    ...
> Technical Debt Item #2
    ...
...
```

Figure 4.1: The DebtFlag mechanism's documentation structure for technical debt (see Publication *P1*) © 2013 IEEE

item-forming elements forwards, and new technical debt states are assessed for software assets, it is highly likely that a single asset will be affected by multiple items through propagation of their DebtFlag elements. These assets contribute valuable research data as they allow for probing Motivation *M5*.

Second, the DebtFlag mechanism defines the DebtFlag elements to have Technical Debt Types (TDT). The types capture the distinct capabilities observable for the debt in this element. For example introduction of non-required variables and deviation from original design are TDTs which can affect a method interface software asset and thus be included in the DebtFlag element associated to the asset. The types also house the propagation rule sets discussed in Section 4.1.1. In defining these types, the publication contributes an avenue for gathering research data. As an organization starts using the DebtFlag mechanism, they will also start to accumulate TDT definitions. Retrospective review into the definitions will provide information regarding what types are used by the organization, in association to which software assets, and with what frequency. Further research on this information may indicate previously overlooked types, and it may suggest characteristic types the organization should enhance management for.

C9: Technical Debt Propagation Characterization Publication *P3* provides further theoretical value in *Examining Technical Debt Accumulation in Software Implementations* and characterizing its propagation. First, the empirical results analyzed in the publication provided additional verification for the research field's previous work on technical debt propagation. The publication established, for the reviewed assets, that the number of incoming dependencies is relational to the number of propagation paths as per Bianchi et al. (2001), and saw that dependency propagation was indeed the driver of technical debt propagation. Further, the observations concurred with McGregor et al. (2012)'s notions of technical debt diminishing due to increasing number of dependency layers: the empirical data showed technical debt instances to not propagate infinitely, but rather the propagation converged after reaching path length of less than five.

Second, the publication observed additional, more intrinsic, technical debt propagation characteristics from the empirical data. Analyzing the technical debt propagation trees, recorded for individual technical debt instances, it was noted that they could be roughly divided into two different shapes. Trees of instances which had their initial cause, the root of the tree, in a software asset that captured a model for the system where shallow and wide in shape. This meant that the technical debt rarely propagated far, but the initial cause had a wide spread, immediate effects on associated assets. Analogously, instances originating from assets housing control logic had propagation trees which were deep and narrow in shape. This meant that the effects of the instance were limited to a smaller area, but penetrated assets, possibly in different architectural layers, much farther. Both, verification of the previously noted characteristics and delivery of new observations, serve the technical debt research, as exhaustive understanding of the control targets, the technical debt instances, is of high importance to management solutions emerging from the research domain.

C10: Technical Debt Propagation's Channel Description Publication *P5*, titled *Modelling Propagation of Technical Debt*, overcomes Research Question *RQ2* in part by defining the technical debt channel. Two details from the definition have further theoretical value. Firstly, the channel is defined as an extension of an information medium. Ability to define the channel as such enables further research fields to be associated to assist studying the channels. For example, looking at the channels from the viewpoint of information theory (Cover and Thomas, 1991) allows arguing for the channels integrity, information content, and entropy to produce further rules for technical debt propagation. Another emerging viewpoint is social-debt (Tamburri et al., 2013) examining the socio-technological decisions of software development. It is possible to appraise and even expand the propagation channels to stakeholders to include social studies as a viewpoint.

Second, introducing the channel as a medium required that the delivered information is described for technical debt. In practice, the publication provided a mapping between the established technical debt properties (see Section 2.4) and the information content. Publication *P5* states that technical debt channel accumulates principal for a technical debt instance in the destination entity if *the software change indicates additional resource consumption* and *the entity hosts the technical debt instance's root cause*. Analogously, interest is accumulated if *the software change indicates additional resource consumption* and *the entity does not host the technical debt instance's root cause*. Further, the realization probability property of technical debt could be explained based on the previous. As the probability captures the chance that further resource consumption is invoked by a technical debt instance, the probability becomes the measure of a technical debt channel existing: if the medium has not yet delivered information which has indicated changes in the destination software asset, the destination asset is not part of the propagation path. The aforedescribed technical debt property mapping to a medium's information content serves further research as it defines the requirements for software asset relationships that facilitate technical debt propagation.

C11: Empirical Verification of Technical Debt Properties To overcome Research Question *RQ2*, Publication *P7* executed *Technical Debt and Agile Software Development Practices and Processes: An Industry Practitioner Survey*. The extensive design of the survey resulted into providing supplementary empirical results with later research value. First, the respondents were queried about their technical debt knowledge via allowing them to first define it and then to indicate if they concurred with the ones frequently applied in academia. The results indicated conformance to the presented definitions which provides future research endeavours with more evidence indicating that they can calibrate their solutions to these definitions. More closely, the survey provided verification for McConnell (2007) definition of technical debt, including intentional and unintentional accumulation, and Brown et al. (2010) definition of technical debt's effects.

Second, in documenting characteristics for concrete technical debt instances, the survey used Kruchten et al. (2012) *"Technical Debt Landscape"* (discussed also as part of Izurieta et al. (2012)) to gather the instances' causes. Again, observed popularity for options in the landscape and scarce use of the self-defined causes indicated that the landscape modelled the technical debt cause-space well. As with the previous verifications, it is important that the survey provided empirical evidence for the landscape. The evidence can be used to further argue for or against newly developed technical debt management solutions. Further, the frequencies indicated for the different causes in the landscape can be seen to focus research endeavours to

certain areas (e.g., *architecture is inadequate* was the most common cause for the recorded instances).

Finally, the survey demonstrated the communication gap (see Section 4.1.1) as dispersed opinions regarding software development practices and processes which involve multiple stakeholders. However, very similar knowledge and position towards technical debt from all stakeholders was also recorded which can be seen to contribute encouragement to research technical debt for bridging the gap.

Practical Aspects

Contributions which have direct application in the software development domain are considered to be of value from the practical aspect. For the publications overcoming the dissertation's primary research questions (see Section 3.1), additively and analogously to the previous section's theoretical contributions, the following exclusively examines the publications for supplementary results of practical value.

C12: Context Independent Technical Debt Management In overcoming Research Question *RQ1* with a tool and guidance for its usage, Publication *P1*, titled *DebtFlag: Technical Debt Management with a Development Environment Integrated Tool*, makes three practical contributions. First, as explained in Sections 3.3.1 and 4.1.2 the DebtFlag mechanism documents TDIs as DebtFlag elements which can have several types (TDTs). The TDTs, like the elements themselves, allow accumulating, organizing, and sharing knowledge within and outside the organization. The knowledge is accumulated when the DebtFlag mechanism is used and new types are declared midst development of software assets. The organization may then organize this knowledge as the types can be refined, combined, or specialized to better reflect the development culture. Finally, the organization may share this knowledge as they can transfer the accumulated type library to new projects or to new organizations. Analogously, the organization may adopt well-defined libraries, for example when starting with new technology contexts, from outside.

Second, overcoming *RQ1* required development of documentation and representation procedures for technical debt, but integration of these as part of IDEs resulted in discovering micro-management for technical debt. Previous research on technical debt management (see Section 2.4) has worked based on explicit decisions being made on documented technical debt instances. Capturing and representing the technical debt persistently and directly on the software assets ensures that the assets developers are fully aware of the debt's presence (see Figure 4.2). This enables the developers to make fine adjustments to their work, either to avoid or to repair technical

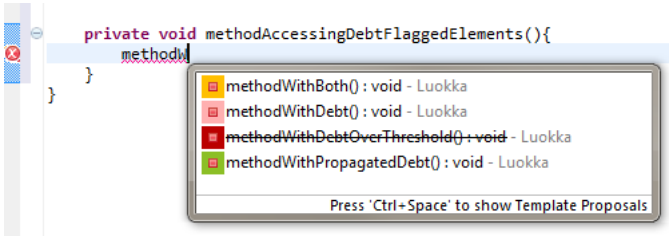


Figure 4.2: The DebtFlag tool modifying the Eclipse IDE's content assist in order to indicate varying levels of technical debt in referable elements (see Publication *P1*) © 2013 IEEE

debt, as efficiently as possible. Noting that software assets can form relations super-linearly which may amplify small technical debt granules into considerable obstacles (see Section 2.5.2), enabling micro-management may have a significant overall effect on the organizations technical debt accumulation.

Finally, third, the DebtFlag provides the first tool-supported implementation of the TDMF. The TDMF is an abstract description of the process and the artifacts required. It can be adopted as is to most software development environments. However, in order to fully adhere to the process description and to ensure that the information gathered, tracked, and used by the process remains up-to-date and coherent, the tool implemented version should be considered as it is not affected by the same error-factors (e.g., subjective implementation). For this reason, a tool implemented approach can be easier to argue for adopting.

C13: Force-Optimal Layout of AST Information For the graph illustration approach presented in Publication *P4*, titled *Illustrating Software Modifiability – Capturing Cohesion and Coupling in a Force-Optimized Graph*, multiple supplementary use-cases can be observed with practical value. First, the presented approach is applicable to any asset relation data which distinguishes asset connection strengths. The data can be automatically generated for all contexts from which Abstract Syntax Trees (AST) can be formed. The publication demonstrated layout for dependency data which directly communicates cohesion and coupling characteristics of the system for the organization. Noteworthy is also that rather than presenting numbers (e.g., average complexity for asset group[1]) the approach provides visual cues, like structure strength in relative distance to other system parts, which acknowledge all assets present in the system.

Second, the sub-graph separation procedure provides the organization with a possibility of tracking the evolution of a particular structure over

[1]The popular SonarQube tool uses a suite of complexity measures: `http://docs.sonarqube.org/display/SONAR/Metric+Definitions`

time. If revision history is available, several graphs can be produce for which the sub-graph separation approach applies path finding in order to identify and re-identify the structure's evolution. Again, it is possible for the organization to exercise the sub-graphing on any data which fits the previous description.

Finally, third, the used force-optimization algorithm, ForceAtlas2, can be used for dynamic representation of asset and asset group relation forming. The ForceAtlas2 algorithm supports non-converging continuous graph layouting. This allows the organization to initially layout a particular system graph, and follow this with introducing further assets into the graph. For example, adding the assets, representing an open-source component that is to be integrated, into the graph representing the current system assets and their structures. The continuous layout will then re-find the force-optimized form of the resulting graph. Observing this dynamic layouting allows practitioners to see how the addition disturbs the pre-existing structures of the initial system.

C14: Technical Debt Propagation Model Library Publication *P5* answered Research Question *RQ3* by *Modelling Propagation of Technical Debt*. The publication discussed the practical value of accumulating a library of technical debt models. First, very similarly to the library of TDIs—the rule sets of which the models are also a part of—accumulated through application of the DebtFlag mechanism, the models may be used for knowledge management and transfer inter- and intra-organizations. For predictive analysis, the models can be used to describe which technical debt propagation paths are enabled for newly created software assets based on which source components match the asset. Similarly, the models may deliver further evidence for resolving problematic software assets by describing the asset as destination, the accumulation point, for all technical debt models for which the asset matches the destination points. Finally, the models may also aid in narrowing the previously discussed communication gap, as they may describe which two contexts, possible though by an organization to be independent, are actually dependent; steering the representatives of these contexts to increase communication.

Finally, it should also be noted that the models describe the bare minimum which is required to capture technical debt propagation in a non project-context dependent manner. The publication notes that if the historical data, form which the models are formed, carries additional information, like effort to implement changes, and it is decomposable as required by the modelling process, it is possible to define the models with this additional data. In case the decomposable effort data is available, the models could be used to provide estimates of technical debt's impact to aid in organizing the assets' development.

C15: In-depth Characterization of Software Organizations Publication *P7*, titled *Technical Debt and Agile Software Development Practices and Processes: An Industry Practitioner Survey*, in meeting Research Question *RQ3* provides supplementary characterizations which have practical potential. First, the publication describes how respondents perceived their adopted set of agile practices and processes to meet the project's management needs in addition to querying if the adopted set was able to cover all matters which should be subjected under management. Mapping to provided background details, an organization can use these results to contrast and reflect on the adequacy of its own management, and possibly derive improvements in adhering or deviating more from the established development characteristics.

Second, in relation to the previous, the publication established the adoption levels for specific agile software development practices and processes. In reflecting its own background details to the adoption levels, an organization, possibly entering a market matching these details, may establish what is the current state-of-practice (for example, the selected set of practices and process for particular organization size, project count, and iteration time). This provides the organization with further details upon which to reflect the choice of a development approach with respect to available software initiatives.

4.2 Limitations

Wohlin et al. (2012) discuss four categories of issues that may affect the validity of an experiment. While general issues were identified for the motivations, applicable validity issue categories are adapted for revising the research questions and their answers. The four categories consist from issues that affect construct validity, internal validity, conclusion validity, and external validity. *Construct validity* deteriorates when the foundations, the constructs, on which the research is established are affected. For example, researcher bias or hypothesis guessing may take place. *Internal validity* is affected if the execution of the research is not ideal. For example, experiment testing is unsuccessful or maturation affects gathered results. *Conclusion validity* is concerned about the methods and reliability with which research outcomes are produced. For example, low statistical power and choice of measures. *External validity* considers the generalizability of the research outcomes to other similar populations and settings. For example, is the set of respondents selected for the study representative of the population intended to be examined.

RQ1 When Publication *P1* answers Research Question *RQ1* (*Can Tooling Support be Provided for Subjective and Context Specific Technical Debt Identification and Estimation?*) it disregards other possible avenues for technical debt management. This can be seen to influence the question's construct validity. Namely, development methods which could introduce changes in applied software practices, processes, and assets, thus inflicting the culture of the particular organization—something that Falessi et al. (2013) note as a requirement for providing tool support for technical debt management. It should be noted that any level of automatization requires computation and thus results into pursuing a tooling approach. Regardless of this, future pursuits should still remain sensitive for other solutions as well; especially when they can be combined with emerging tooling approaches. Further, the tool produced in Publication *P1* is an IDE plug-in which affects external validity: while the DebtFlag mechanism is detachable from the tool implementation, the publication in question does not explore the opportunity of realizing the mechanism in environments wherein IDEs are not used.

RQ2 The uniqueness aspect discussed as an issue for Motivations *M4* and *M5* can be seen to affect Research Question *RQ2* (*How Can Technical Debt Propagation be Captured and Documented as Models?*) as well. In answering *RQ2* by way of comprising the technical debt models through a similarity-comparison-procedure, Publication *P5* imposes generalization on to the technical debt instances. This is a cause for internal validity issues: there is a danger that the generalization will abstract out the unique properties, the modelling approach should support a level of composition that allows for extracting both the more common and thus more general components as one model and the possibly unique attributes as their own components; equal to other models. The modelling approach introduced in Publication *P5* accounts for this—but not explicitly—as no propagation characteristics are disregarded in the abstraction process. Further, Publication *P5* tests the modelling process on a limited data set which can be seen to affect the conclusion validity for the research question. While the channel description is technology context independent and thus generally applicable, the publication in question also notes this limitation. Preceding Publication *P2* documents a number of similar technical debt propagation cases which increase the dependability of the results slightly, but other technology contexts should be further explored; Publication *P6* works towards meeting this goal with mining software repositories to identify further cases.

RQ3 The Research Question *RQ3* (*What Characteristics do Software Development Environments and their Technical Debt Possess?*) is implemented primarily to facilitate integration of the technical debt management tooling

approach produced for the previous research questions. As such, focus is given to characteristics of technical debt instances in addition to technical debt sensitivity and adoption level of different software development practices and processes that could accommodate the tool. There are, however, additional viewpoints which can be seen to produce further characteristics. Publication *P7* proceeds to survey the question, but excludes these viewpoints which raises internal validity concerns. The viewpoints include non-environmental characteristics like the previously discussed social interactions as well as the effect of time on previously identified characteristics; namely organization wide learning. Initial efforts towards filling this characterization deficit can be seen made by Publication *P6* as it describes observing development velocity factors as a function of time in environments wherein periodic technical debt inducers can be identified (in case of *P6* changes to adopted 3rd party APIs). As the publication identifies their importance, they are named as future work objective for the particular research project.

4.3 Future Work

In the grand scheme of things, the future work for this dissertation encompasses realization of *efficient technical debt management with tailor-integrated (RQ3) technical debt identification and estimation tool and process (RQ1) that supports propagation model based automatic information maintenance (RQ2)*, as Section 3.1 establishes. While the previous extensively reviews the contributions made towards this goal, further efforts must be made in order to fully achieve it.

Firstly, a library of technical debt propagation models is required in order to capture the debt's capabilities for the most prominent contexts and combinations of contexts. This can be achieved by applying the model forming process introduced in Publication *P5* for the pre-screened, technical debt sensitive projects, accumulated by Publication *P6* via applying MSR to open-source project hubs. Whilst the models carry value in themselves (see Section 4.1.2), their primary user will be the DebtFlag mechanism that, via operationalization of the models, provides automatic technical debt information updates for the modelled contexts.

The DebtFlag mechanism must be extended to cover a wider range of development environments so that it can reach the development of as many software assets as possible. This extension can be done in parallel to accumulating the propagation model library. The design of the DebtFlag mechanism, and its first implementation, in Publication *P1* accommodates the extension: the core architecture of the DebtFlag tool allows new AST (Abstract Syntax Tree) parsers to be attached, meaning that only the language-specific parser must be re-implemented to make the tool support a new

language within the Eclipse IDE. Further, it has been considered that the plug-in would also be redeveloped for other IDEs, namely the JetBrains tool set[2], so as to not limit developers environment requirements and to support further programming and modelling language contexts. The second part of the tool, the web-interface, is by design able to accommodate multiple plug-ins which communicate with the web-interface only via a shared data structure and a database.

Finally, when the DebtFlag mechanism has been extended to a viable tool suite, supporting multiple technology contexts and development mechanisms like revision control systems, the accumulated model library will be attached to the tool, as described in Publication *P2*'s third objective. This will allow the DebtFlag mechanism to reach its full potential: to be able to extensively capture subjective opinions about technical debt, to provide automated information updates based on observing the evolution for the hosting software assets, and to serve this information both immediate to the software assets for micro-management and as an up-to-date technical debt list for the project management.

The fully functional DebtFlag suite will then be used as the independent variable in a research experiment. Publication *P7*'s findings regarding software practices and processes sensitive to technical debt as well as the characteristics of concrete technical debt instances will respectively be used as the dependent variables and for fine-tuning the tool for integration. The experiment will then be executed in order to demonstrate that *enhanced technical debt management has a positive effect on the efficiency and sustainability of a software development project.*

[2] `https://www.jetbrains.com/products.html`

Bibliography

Alves, N. S., Ribeiro, L. F., Caires, V., Mendes, T. S., and Spínola, R. O. (2014). Towards an ontology of terms on technical debt. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 1–7. IEEE.

Avgeriou, P., Kruchten, P., Ozkaya, I., and Seaman, C. (2016). Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports*, 6(4):110–138.

Bansiya, J. and Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering*, 28(1):4–17.

Basili, V. R. (1992). Software modeling and measurement: the Goal/Question/Metric paradigm. *Technical Report Produced at University of Maryland*.

Basili, V. R. (1993). The experimental paradigm in software engineering. In *Experimental Software Engineering Issues: Critical Assessment and Future Directions*, pages 1–12. Springer.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al. (2001). Manifesto for Agile Software Development. Online Publication. `http://agilemanifesto.org/`.

Bennett, K. (1995). Legacy systems: Coping with success. *IEEE software*, 12(1):19–23.

Bianchi, A., Caivano, D., Lanubile, F., and Visaggio, G. (2001). Evaluating software degradation through entropy. In *Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International*, pages 210–219. IEEE.

Booth, W. C. (1978). Metaphor as rhetoric: The problem of evaluation. *Critical Inquiry*, 5(1):49–72.

Bossavit, L. (2012). *The Leprechauns of Software Engineering*. Leanpub.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., et al. (2010). Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM.

Bruegge, B. and Dutoit, A. H. (2004). *Object-Oriented Software Engineering Using UML, Patterns and Java-(Required)*. Prentice Hall.

Budde, R., Kautz, K., Kuhlenkamp, K., and Züllighoven, H. (1992). Prototyping. In *Prototyping*, pages 33–46. Springer.

Buschmann, F. (2011). To pay or not to pay technical debt. *IEEE software*, 28(6):29–31.

Cover, T. M. and Thomas, J. A. (1991). *Elements of Information Theory*. John Wiley & Sons.

Davis, A. M., Bersoff, E. H., and Comer, E. R. (1988). A strategy for comparing alternative software development life cycle models. *IEEE Transactions on Software Engineering*, 14(10):1453–1461.

Dayani-Fard, H. and al, e. (1999). *Legacy Software Systems: Issues, Progress and Challenges*. IBM: Technical Report TR-74.165-k.

Dybå, T. and Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and software technology*, 50(9):833–859.

Eisenberg, R. J. (2012). A threshold based approach to technical debt. *ACM SIGSOFT Software Engineering Notes*, 37(2):1–6.

Falessi, D., Shaw, M. A., Shull, F., Mullen, K., and Keymind, M. S. (2013). Practical considerations, challenges, and requirements of tool-support for managing technical debt. In *Managing Technical Debt (MTD), 2013 4th International Workshop on*, pages 16–19. IEEE.

Fayad, M. E., Laitinen, M., and Ward, R. P. (2000). Thinking objectively: software engineering in the small. *Communications of the ACM*, 43(3):115–118.

Feathers, M. (2004). *Working effectively with legacy code*. Prentice Hall Professional.

Fowler, M. (2009). Technicaldebtquadrant. Online Publication. `http://martinfowler.com/bliki/TechnicalDebtQuadrant.html`.

Fowler, M. and Beck, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Fowler, M., Beck, K., and Brant, J. (1999). Refactoring: improving the design of existing code. *Refactoring: Improving the Design of Existing Code*.

Gildea, P. and Glucksberg, S. (1983). On understanding metaphor: The role of context. *Journal of Verbal Learning and Verbal Behavior*, 22(5):577–590.

Griffith, I., Reimanis, D., Izurieta, C., Codabux, Z., Deo, A., and Williams, B. (2014). The correspondence between software quality models and technical debt estimation approaches. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 19–26. IEEE.

Guo, Y. and Seaman, C. (2011). A portfolio approach to technical debt management. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 31–34. ACM.

Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Da Silva, F., Santos, A., and Siebra, C. (2011a). Tracking technical debt - An exploratory case study. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 528–531. IEEE.

Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Da Silva, F. Q., Santos, A. L., and Siebra, C. (2011b). Tracking technical debt—An exploratory case study. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 528–531. IEEE.

Ho, T. T. and Ruhe, G. (2014). When-to-release decisions in consideration of technical debt. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 31–34. IEEE.

Holvitie, J. (2014). Software Implementation Knowledge Management with Technical Debt and Network Analysis. In *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on*, pages 1–6. IEEE.

Holvitie, J. and Leppänen, V. (2013). DebtFlag: Technical Debt Management with a Development Environment Integrated Tool. In *Proceedings of the 4th International Workshop on Managing Technical Debt (MTD)*, pages 20–27. IEEE.

Holvitie, J. and Leppänen, V. (2014). Illustrating Software Modifiability–Capturing Cohesion and Coupling in a Force-Optimized Graph. In *Com-*

puter and Information Technology (CIT), 2014 IEEE International Conference on, pages 226–233. IEEE.

Holvitie, J. and Leppänen, V. (2015). Examining Technical Debt Accumulation in Software Implementations. *International Journal of Software Engineering and Its Applications*, 9(6):109–124.

Holvitie, J., Leppänen, V., and Hyrynsalmi, S. (2014). Technical Debt and the Effect of Agile Software Development Practices on It–An Industry Practitioner Survey. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 35–42. IEEE.

Holvitie, J., Licorish, S., Martini, A., and Leppänen, V. (2016a). Co-Existence of the 'Technical Debt' and 'Software Legacy' Concepts. In *Technical Debt Analytics, First International Workshop on*. CEUR-WS.

Holvitie, J., Licorish, S., Spinola, R., Hyrynsalmi, S., Buchan, J., Mendes, T., MacDonnell, S., and Leppänen, V. (2017). Technical Debt and Agile Software Development Practices and Processes: An Industry Practitioner Survey. *Journal article under review.*

Holvitie, J., Licorish, S. A., and Leppänen, V. (2016b). Modelling Propagation of Technical Debt. In *Software Engineering and Advanced Applications (SEAA), 2016 42th Euromicro Conference on*, pages 54–58. IEEE.

Izurieta, C., Vetrò, A., Zazworka, N., Cai, Y., Seaman, C., and Shull, F. (2012). Organizing the technical debt landscape. In *Proceedings of the Third International Workshop on Managing Technical Debt*, pages 23–26. IEEE Press.

Kagdi, H., Collard, M. L., and Maletic, J. I. (2007). 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*, 19(2):77–131.

Klinger, T., Tarr, P., Wagstrom, P., and Williams, C. (2011). An enterprise perspective on technical debt. In *Proceedings of the 2nd Workshop on managing technical debt*, pages 35–38. ACM.

Knuth, D. E. (1998). *The art of computer programming: sorting and searching*, volume 3. Pearson Education.

Kruchten, P., Nord, R. L., and Ozkaya, I. (2012). Technical Debt: From Metaphor to Theory and Practice. *Ieee software*, 29(6).

Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183.

Larman, C. and Basili, V. R. (2003). Iterative and incremental development: A brief history. *IEEE Computer Society*, pages 47–56.

Letouzey, J.-L. and Ilkiewicz, M. (2012). Managing technical debt with the sqale method. *IEEE software*, 29(6):44–51.

Li, Z., Avgeriou, P., and Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220.

Li, Z., Liang, P., Avgeriou, P., Guelfi, N., and Ampatzoglou, A. (2014). An empirical investigation of modularity metrics for indicating architectural technical debt. In *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*, pages 119–128. ACM.

Licorish, S., Holvitie, J., Spinola, R., Hyrynsalmi, S., Buchan, J., Mendes, T., MacDonnell, S., and Leppänen, V. (2016). Adoption and Suitability of Software Development Methods and Practices. In *2016 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE.

Marinescu, R. (2012). Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5):9–1.

McConnell, S. (2007). Technical debt. Online Publication. 10x Software Development Blog,(Nov 2007). Construx Conversations. URL= `http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx`.

McGregor, J. D., Monteith, J. Y., and Zhang, J. (2012). Technical debt aggregation in ecosystems. In *Managing Technical Debt (MTD), 2012 Third International Workshop on*, pages 27–30. IEEE.

Morgenthaler, J. D., Gridnev, M., Sauciuc, R., and Bhansali, S. (2012). Searching for build debt: Experiences managing technical debt at Google. In *Proceedings of the Third International Workshop on Managing Technical Debt*, pages 1–6. IEEE Press.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The art of software testing*. John Wiley & Sons.

Nugroho, A., Visser, J., and Kuipers, T. (2011). An empirical model of technical debt and interest. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 1–8. ACM.

Parnas, D. L. and Weiss, D. M. (1985). Active design reviews: principles and practices. In *Proceedings of the 8th international conference on Software engineering*, pages 132–136. IEEE Computer Society Press.

Paternoster, N., Giardino, C., Unterkalmsteiner, M., Gorschek, T., and Abrahamsson, P. (2014). Software development in startup companies: A systematic mapping study. *Information and Software Technology*, 56(10):1200–1218.

Peffers, K., Tuunanen, T., Rothenberger, M. A., and Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77.

Pulkkinen, P., Holvitie, J., Nevalainen, O. S., and Leppänen, V. (2015). Reusability Based Program Clone Detection: Case Study on Large Scale Healthcare Software System. In *Proceedings of the 16th International Conference on Computer Systems and Technologies*, pages 90–97. ACM.

Rahikkala, J., Leppänen, V., Ruohonen, J., and Holvitie, J. (2015). Top Management Support in Software Cost Estimation: A Study of Attitudes and Practice in Finland. *International Journal of Managing Projects in Business*, 8(3):513–532.

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

Robson, C. (2002). Real world research. 2nd. *Edition. Blackwell Publishing. Malden.*

Royce, W. W. (1970). Managing the development of large software systems. In *proceedings of IEEE WESCON*, pages 328–338. Los Angeles.

Runeson, P. and Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164.

Schmid, K. (2013). A formal approach to technical debt decision making. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, pages 153–162. ACM.

Schwaber, K. and Beedle, M. (2002). *Agile Software Development with Scrum.* Pearson.

Seaman, C. and Guo, Y. (2011). Measuring and monitoring technical debt. *Advances in Computers*, 82:25–46.

Seaman, C., Guo, Y., Izurieta, C., Cai, Y., Zazworka, N., Shull, F., and Vetrò, A. (2012). Using technical debt data in decision making: Potential decision approaches. In *Proceedings of the Third International Workshop on Managing Technical Debt*, pages 45–48. IEEE Press.

Selby, R. W. (2005). Enabling reuse-based software development of large-scale systems. *IEEE Transactions on Software Engineering*, 31(6):495–510.

Shull, F., Falessi, D., Seaman, C., Diep, M., and Layman, L. (2013). Technical debt: Showing the way for better transfer of empirical results. In *Perspectives on the Future of Software Engineering*, pages 179–190. Springer.

Singh, V., Snipes, W., and Kraft, N. A. (2014). A framework for estimating interest on technical debt by monitoring developer activity related to code comprehension. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 27–30. IEEE.

Slonneger, K. and Kurtz, B. L. (1995). *Formal syntax and semantics of programming languages*, volume 340. Addison-Wesley Reading.

Sommerville, I. (2008). *Software Engineering. International computer science series*. Addison Wesley.

Stavru, S. (2014). A critical examination of recent industrial surveys on agile method usage. *Journal of Systems and Software*, 94:87–97.

Suovuo, T., Holvitie, J., Smed, J., and Leppänen, V. (2015). Mining Knowledge on Technical Debt Propagation. In *Programming Languages and Software Tools (SPLST), 2015 Symposium on*, pages 281–295. CEUR-WS.

Suryanarayana, G., Samarthyam, G., and Sharma, T. (2014). *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann.

Sutton, S. M. (2000). The role of process in a software start-up. *IEEE Software*, 17(4):33.

Tamburri, D. A., Curtis, B., Fraser, S. D., Goldman, A., Holvitie, J., da Silva, F. Q. B., and Snipes, W. (2016). Social Debt in Software Engineering: Towards a Crisper Definition. *Dagstuhl Reports*, 6(4).

Tamburri, D. A., Kruchten, P., Lago, P., and van Vliet, H. (2013). What is social debt in software engineering? In *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*, pages 93–96. IEEE.

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

Trochim, W. M. K. (2016). Deduction & induction. Online Publication. `http://www.socialresearchmethods.net/kb/dedind.php`.

Tsantalis, N. and Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367.

Walfish, S., McAlister, B., O'Donnell, P., and Lambert, M. J. (2012). An investigation of self-assessment bias in mental health providers. *Psychological Reports*, 110(2):639.

Whitehead, J. and McNiff, J. (2006). *Action research: Living theory*. Sage.

Wiklund, K., Eldh, S., Sundmark, D., and Lundqvist, K. (2012). Technical debt in test automation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 887–892. IEEE.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Zazworka, N., Shaw, M. A., Shull, F., and Seaman, C. (2011). Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 17–23. ACM.

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

Zhu, H., Zhou, M., and Seguin, P. (2006). Supporting software development with roles. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 36(6):1110–1123.

Zmud, R. W. (1980). Management of large software development efforts. *MIS quarterly*, pages 45–55.

Paper I

DebtFlag: Technical Debt Management with a Development Environment Integrated Tool

Holvitie, Johannes and Leppänen, Ville (2013). In *Proceedings of the 4th International Workshop on Managing Technical Debt (MTD)*, pages 20–27. IEEE

DebtFlag: Technical Debt Management with a Development Environment Integrated Tool

Johannes Holvitie, Ville Leppänen
TUCS - Turku Centre for Computer Science
&
Department of Information Technology
University of Turku
Turku, Finland
{jjholv,ville.leppanen}@utu.fi

***Abstract*—In this paper, we introduce the *DebtFlag* tool for capturing, tracking and resolving technical debt in software projects. *DebtFlag* integrates into the development environment and provides developers with lightweight documentation tools to capture technical debt and link them to corresponding parts in the implementation. During continued development these links are used to create propagation paths for the documented debt. This allows for an up-to-date and accurate presentation of technical debt to be upheld, which enables developer conducted implementation-level micromanagement as well as higher level technical debt management.**

***Index Terms*—Technical debt, technical debt management, source code assessment, source code analysis, DebtFlag.**

I. Introduction

Ward Cunningham was the first to coin the term technical debt [1]. In his technical report deviation from the design incurs the principal of technical debt. Refactorization pays it back. Development on top of the principal counts as technical debt's interest and hindered development constitutes as paying interest. Like with its financial counterpart: technical debt is acceptable as long as its payback is managed.

Technical debt management is a rather new research area, interested in introducing control over technical debt by providing projects with means to identify, assess and payback technical debt. Seaman et al. [2] raise the availability and clarity of technical debt information as one of the key factors to successful technical debt management.

Highly complex, self-emergent and frequently changing software products are a challenging ground for technical debt information production. Automatic approaches are capable of accommodating the change rates that major development projects introduce, but their reliance onto statically predefinable models make them incapable of modeling the entire requirement space [3], [4]. Manual approaches capture the entire space [5], [6] but due to their nature they consume a large amount of development resources which disallows their frequent use.

The mechanism we introduce has been designed to exploit the benefits of both aforementioned assessment approaches. *DebtFlag* links structured observations about technical debt to related parts in the software implementation. The structure does not limit the declaration but it retains an equivalency between the entries that makes automatic updates possible.

When used in software projects, *DebtFlag* captures technical debt through lightweight documentation tools that integrate into the development environment. It tracks the propagation of technical debt by building dependency trees for associated software implementation parts. *DebtFlag* allows to resolve technical debt by supporting its management on two different levels. Developer conducted micromanagement through maintaining an implementation level representation of technical debt and project level management by making *DebtFlag* cater for the information needs of higher level approaches.

II. *DebtFlag* Mechanism

The *DebtFlag* mechanism has been designed to be compatible with different software implementation techniques. This section provides a description for it, explaining the documentation structure for technical debt, the requirements and functionality for automation and describes how these pursue two different technical debt management approaches.

A. Structure of Documented Technical Debt

The structure for documenting technical debt with the *DebtFlag* mechanism is based onto the documentation structure introduced as part of the Technical Debt Management Framework (TDMF) [7], [8] by Seaman et al.. This structure is extended in the *DebtFlag* mechanism in order to decompose entries into reusable components as well as to properly present technical debt at the implementation level.

The Technical Debt Management Framework is a three parted approach on managing technical debt in software projects. It relies onto a Technical Debt List (TDL) constructed in the first, technical debt identification, part. The list is populated with Technical Debt Items (TDI), which correspond to single atomic occurrences of technical debt in the project.

A Technical Debt Item documents and upholds a set of information [7]. A description explains the debt's type, location and reasoning for its acquirance. An estimate for the debt's principal indicates how much resources are required to pay it back – to make this partition fully adhere to the design. While,

978-1-4673-6443-0/13/$31.00 © 2013 IEEE

an interest estimates the probability and amount of extra work this principal can cause to future development.

```
Technical Debt List
> Technical Debt Item #1
    > DebtFlag #1
        - Time and Date
        - Author
        - Location
        - Description
        - Type Declaration
            > Technical Debt Type #1
                - Name
                - Description
                - Context
                - Propagation rules
            > Technical Debt Type 2
                ...
    > DebtFlag #2
        ...
    ...
> Technical Debt Item #2
    ...
...
```

Fig. 1. The *DebtFlag* mechanism's documentation structure for technical debt

This structure is extended with an additional level. In the *DebtFlag* mechanism a TDI can consist out of a single or multiple *DebtFlag* elements. A *DebtFlag* element is a link between a technical debt observation and an implementation part defined by the technique. For example a package, a class or a method in object-oriented technologies.

By allowing TDIs to be constructed with *DebtFlag* elements we want to preserve a degree of freedom: the description of a TDI's location is not limited to a single area predefined by the implementation technique, but rather it can encompass an unlimited amount and combination of them. This makes it possible for a single TDI to have unrestricted propagation capabilities. At the same time, the possibility to use *DebtFlag* elements as equals to TDIs is kept.

The attributes of a *DebtFlag* element are inherited from a TDI and consist out of a time and a date, an author, a type, a location and a description. The time and date indicate when the observation was made. The author corresponds to the responsible developer. The location to an element in the implementation, as described by the previous paragraph. The type attribute has been extended and made component based.

The *DebtFlag* element type consists out of a set of technical debt types. These types can be either predefined or created during use. A technical debt type documents a name, a description, a context and a propagation rule set for it. The type name and the description are self-explanatory. The context attribute binds this type to a certain implementation context - for example a programming language. The propagation rule set is used for declaring the propagation capabilities for this type of technical debt, its principal and the effect the propagation has onto the accumulation of interest.

B. Automation of Technical Debt Propagation

Dynamic functionality of the *DebtFlag* mechanism relies onto being able to automate two processes. The process of identifying source points for the propagation of technical debt and the process of propagating the technical debt according to rule sets and dependencies in the implementation.

Capturing technical debt with the *DebtFlag* mechanism corresponds to creating TDIs by forming collections of *DebtFlag* elements. As stated, a *DebtFlag* element represents a link between a technical debt observation and a corresponding implementation part. Depending onto the used implementation technique, an implementation part can contain several points capable of forming dependencies. Dependencies carry technical debt and increase its interest [9]. Projection of technical debt onto the implementation is thus dependent onto being able to identify those source components responsible for propagating technical debt. This functionality is dependent onto information about the used implementation technique.

After acquiring the source implementation components for technical debt, the *DebtFlag* mechanism completes the projection by propagating technical debt through dependencies while following a possible rule set. The process takes a source component and goes through the *DebtFlag* elements associated with it. For each *DebtFlag* element the technical debt type declared for it determines the rules for its propagation. According to these the *DebtFlag* mechanism associates the *DebtFlag* elements with implementation components that are directly or indirectly dependent onto their source components.

Figure 2 presents a simplified example scenario and how the aforementioned two processes function. Classes `A` and `B` both contain two methods `A.a` and `A.b` as well as `B.c` and `B.d` respectively. Method `B.d` is dependent onto method `B.c` which again is dependent onto method `A.a`. Two *DebtFlag* elements are created. First one for the entire class `A` and a second one for just the method `B.c`. From the aforementioned

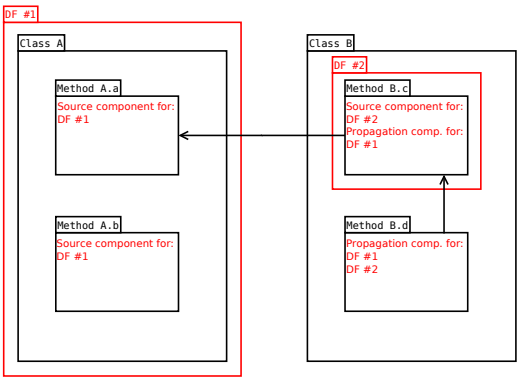


Fig. 2. Propagation of technical debt in the *DebtFlag* mechanism

processes, the first one now returns methods `A.a` and `A.b` as the source points for the first *DebtFlag* element and method `B.c` for the second element. The second process can now be called with these three source points. For method `A.a` it will return a dependency tree containing methods `B.c` and `B.d`, for method `A.b` an empty tree and for method `B.c` a dependency tree containing method `B.d`. Based onto this information and the propagation rules associated with each *DebtFlag* element, the *DebtFlag* mechanism associates each component in Figure 2 with captured technical debt.

C. Technical Debt Management

The *DebtFlag* mechanism is designed to support technical debt management in two different forms. Project and organization level management through supporting the TDMF and implementation level micromanagement by decomposing and projecting technical debt onto the implementation.

The Technical Debt Management Framework allows a variety of processes to be used for managing technical debt. The functionality of the TDMF is dependent onto the existence of the Technical Debt List. The *DebtFlag* mechanism has been designed so as to be able to efficiently construct and maintain the TDL.

The TDL is populated with Technical Debt Items that are formed from *DebtFlag* elements. The *DebtFlag* elements inherit their basic attributes from the TDI and thus answer to them as a set. In order to maintain the TDL during continued development the *DebtFlag* elements are designed not to prompt estimates about technical debt's principal and interest. Rather, these estimates are based on the current number of *DebtFlag* elements forming a TDI, the technical debt types they are associated with and the information about their propagation. Essentially, the used propagation rule set defines the technical interest for all technical debt items. We discuss the importance of this in Section IV-C and how we intend to take this into account in Section V-A.

Micromanagement of technical debt is supported by creating and maintaining a presentation for it on the implementation level. As technical debt is decomposed into source propagation points and propagated onwards according to monitored dependencies the produced information enables the debt to have a presentation at these points. By using for example visualization and restriction (see Section III) the developer can be made aware of otherwise unnoticeable technical debt and its properties. We discuss our expectations for technical debt aware software development in Section IV.

III. *DebtFlag* Tool

The first implementation of the *DebtFlag* mechanism was designed to support the Java programming language. The *DebtFlag* tool is a two parted system consisting out of a plug-in for the Eclipse Integrated Development Environment (IDE) [10] and a separate web application. The *DebtFlag* plug-in is responsible for capturing technical debt through the development environment, tracking its propagation and supporting the micromanagement approach. The web application provides a dynamic presentation of the Technical Debt List compiled from information produced with the *DebtFlag* plug-ins.

A. DebtFlag Plug-In

Eclipse is a popular IDE used especially for developing in the Java language. It is built on the concept of plug-ins and by implementing the first part of the *DebtFlag* tool as such, we are capable of integrating the documentation tools into the development environment, identifying the source implementation elements for technical debt, propagating the debt according to dependencies and building a representation of technical debt on the implementation level.

1) Capturing Technical Debt: Capturing technical debt using the *DebtFlag* plug-in is started by interacting with a Java element. Eclipse provides multiple views to the Java element hierarchies it is used to modify. Valid Java elements from the *DebtFlag* plug-ins perspective range from a package to a class member, such as a method or a global variable. All of these can serve as the location for a *DebtFlag* element and trigger the documentation process.

Figure 3 depicts the listing dialog triggered by interaction with a Java element. In this case, the element is method `c` from class `B` and the dialog displays two *DebtFlag* element listings for it. The first one contains those instances where the

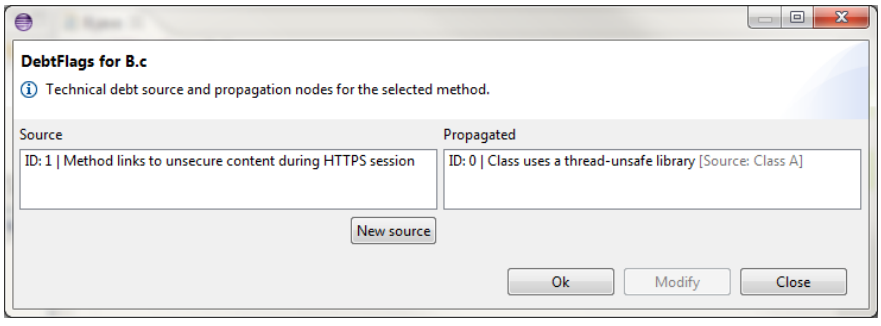


Fig. 3. The *DebtFlag* element list triggered through interaction with a Java element

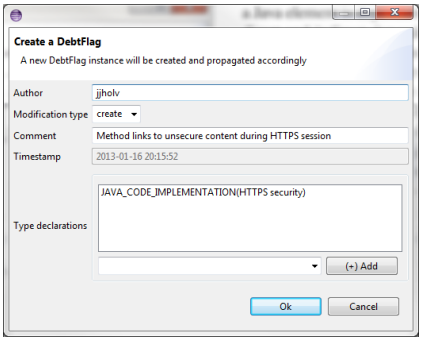


Fig. 4. The creation of a new *DebtFlag* element event

debt's source location is the current Java element, while the second one contains those propagated to it via dependencies. From here the user can choose to either create a new *DebtFlag* element - having this Java element as its source - or to modify any of the displayed *DebtFlag* elements.

Events forming the life span of a *DebtFlag* element are predetermined. It starts with a *create* event, followed by any number of *modify* events and ends with a *resolve* event. Figure 4 depicts the dialog for producing *DebtFlag* element events. In this case, the "create" event of a new *DebtFlag* element for the aforementioned method `B.c`.

The *DebtFlag* element event dialog (see Figure 4) captures the author, the modification event type, the comment, the time and the type declaration for a *DebtFlag* element instance. The location information is not prompted as this corresponds to the Java element triggering this dialog. The author, the modification event type and time attribute are prefilled according to system wide information. The type declaration part shows the currently selected technical debt types used for defining this *DebtFlag*, the accumulated library of available types and the possibility to add new ones. The comment attribute is a free text form intended for providing the reasoning for the event.

As mentioned in Section II-A, a *DebtFlag* element's type is a collection of technical debt types. When defining a new *DebtFlag* element event, if the provided library does not contain a suitable type combination, new ones can be created. Figure 5 depicts the dialog for creating a new technical debt type. It captures the type's name, description, propagation capabilities, context and threshold. Currently, the propagation capability of a type is a binary option for either declaring that this type can propagate through dependencies – increasing the debt's interest – or that it is confined inside the Java element. The threshold attribute is used to communicate the severity of the type by declaring how many dependencies can be formed to elements carrying it before additional measures are enforced (see Section III-A2). A lower threshold can indicate a high principal, rapidly growing interest, probable realization or a combination of these.

2) Implementation Level Representation of Technical Debt: The *DebtFlag* plug-in builds an implementation level representation of technical debt in order to support its micromanagement. The representation uses visualization and restriction in the Eclipse IDE for indicating the presence of technical debt. The information, the representation is based on, is produced with the decomposition and propagation processes described in Section II-B and implemented using the Eclipse Java Development Tools (JDT). The JDT is a core Eclipse plug-in which generates information about Java implementation structures.

The *DebtFlag* plug-in modifies the visual appearance of each Java element that has either source or propagated debt. The effect technical debt has on the visual representation of a Java element is dependent onto two matters. The number of direct and indirect dependencies coming to an element and the technical debt types associated to this element. The end results lead to four representation categories for technical debt. Each category excludes the others and has priority over those mentioned before it.

The *source* category indicates that a Java element is the source point for technical debt propagation (default illustration with light red color). The *propagated* category indicates that a Java element is on the propagation path of technical debt – that it is directly or indirectly dependent to a *source* Java element (default illustration with light green color). The *source and propagated* category combines the former groups (default illustration with orange color). Finally, the *debt over threshold* category is used to indicate that the number of dependencies to this Java element exceeds a threshold defined for its technical debt (default illustration with dark red color).

Figure 6 shows the structure from Figure 2 implemented in Java using the Eclipse IDE while employing the *DebtFlag* plug-in to make two technical debt declarations. The first one is made for class `A` with a technical debt type having a threshold value of one (1). This has resolved into two source points for propagation: methods `A.a` and `A.b`. The second declaration has been made directly for method `B.c` with a technical debt type having a threshold value of two (2).

In Figure 6 method `B.d` is dependent onto method `B.c` which again is dependent onto method `A.a`. As this exceeds the threshold value of `A.a` the *DebtFlag* plug-in uses the *debt*

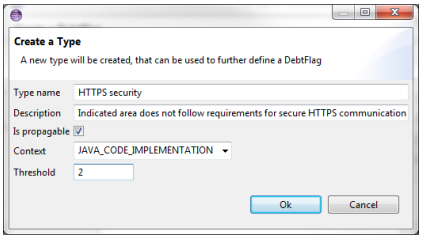


Fig. 5. The creation of a new *DebtFlag* element type

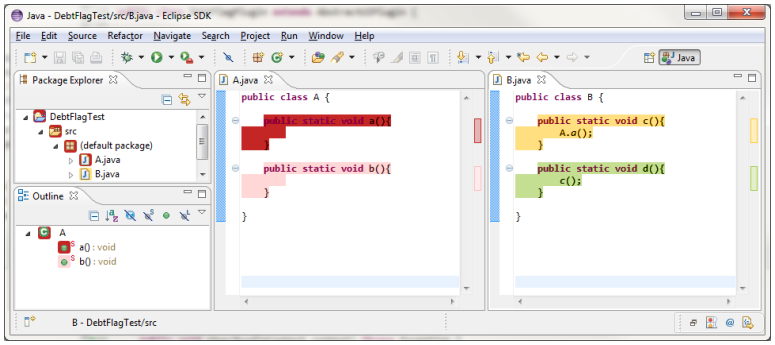


Fig. 6. The main view of the Eclipse IDE, while using the *DebtFlag* plug-in to manage technical debt instances

over threshold category in its visualization. From the same dependency chain, method `B.c` also had its own technical debt declared for it leading to having the *source and propagated* visualization, while method `B.d` only carries propagated debt and hence has the *propagated* visualization. Finally, method `A.b` as the other source point for propagating technical debt in class `A` has the visual appearance of the *source* category.

The other component in forming the representation is restriction. Figure 7 shows the Eclipse content-assist. It provides developers with dynamic content assistance depending onto the cursor's position in the editor. The figure in question shows the content assistant opened when the cursor is inside class `A` (see Figure 6). The restriction is applied here in the form of a strike through over the method `A.a`. This optional feature of the *DebtFlag* plug-in ensures that no new dependencies are introduced for elements that have their threshold crossed by disallowing their use.

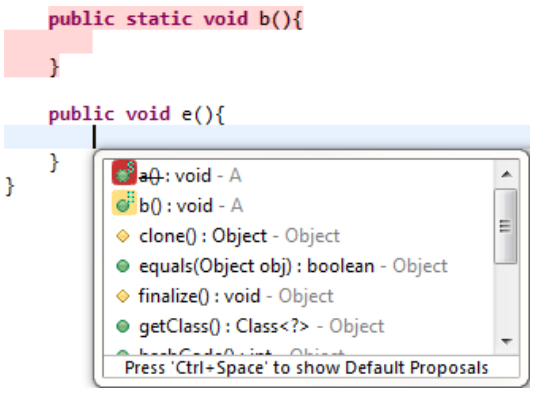


Fig. 7. The Eclipse content-assist, affected by the *DebtFlag* plug-in

Each *DebtFlag* plug-in works on its own set of *DebtFlag* elements. This set corresponds to elements associated with the latest version checked out to the Eclipse IDE and the additions and modifications made to them afterwards. When changes to the implementation are committed through the version control system, the *DebtFlag* plug-in communicates with the Eclipse Team plug-in. This allows the *DebtFlag* plug-in to gain information about possible conflicts between implementation versions and their resolutions. According to this information an individual *DebtFlag* plug-in builds a *DebtFlag* element set that corresponds to the new version and inserts it to a database.

B. DebtFlag Web Application

The *DebtFlag* web application has been created using the Vaadin [11] web application framework. The Vaadin data binding mechanism has allowed us to create a simple and dynamic representation of the *DebtFlag* database. This representation corresponds to the Technical Debt List.

Figure 8 depicts the *DebtFlag* web application. The header bar contains the main controls. From here it is possible to select the project and a version for which the TDL is constructed. The main content changes according to these choices and is two parted. The left hand side part contains the actual TDL. The TDL representation follows the documentation structure presented in Section II-A and it is colored according to the representation categories described in Section III-A2.

The right hand side of the main content of the *DebtFlag* web application (see Figure 8) contains detailed information for a selected *DebtFlag* element. Here the upper partition contains the attributes described in Section II-A and the lower partition contains the dependency tree. The dependency tree has the implementation elements decomposed from the *DebtFlag* element's location as its source nodes and it branches according to the rules defined by the *DebtFlag* element's types. From here, it is easy to see the reasoning for why a threshold value of a particular *DebtFlag* element was crossed.

IV. Discussion

This section covers applying the *DebtFlag* mechanism into software development. The discussion is started by establishing the mechanism's role in a software development environ-

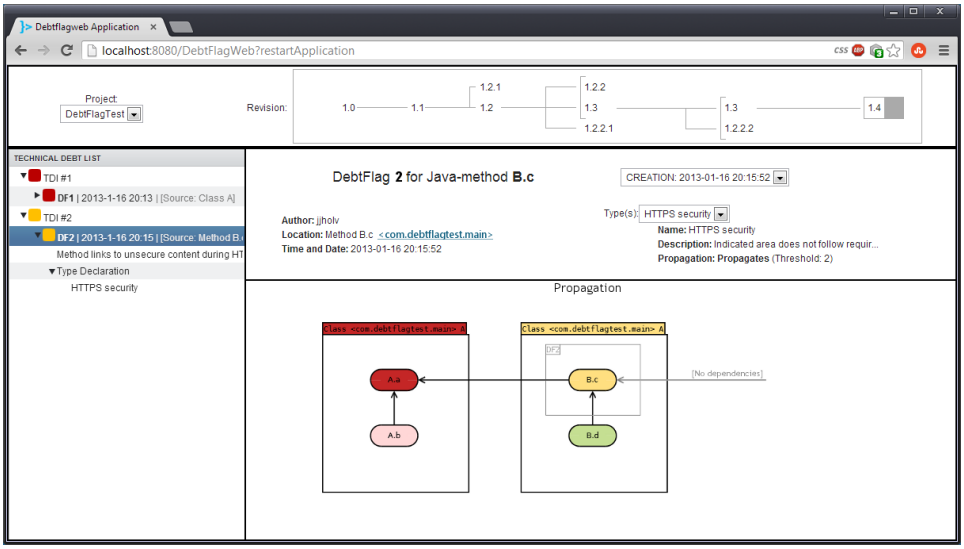


Fig. 8. The *DebtFlag* tool's web interface

ment followed by sections discussing the expectations set for the first implementation of the mechanism. The expectations are made from a software project's perspective and cover foreseeable benefits and challenges.

A. Application in Software Development

The *DebtFlag* mechanism builds the documentation for a software product's technical debt by capturing and processing relevant observations. Capturing observations requires that the mechanism is made available in software development components where observations about the software product's state are made. These observations are processed into a TDL and automatically maintained by the mechanism. The produced technical debt documentation serves as the integration point for further technical debt management approaches and provides information to existing software development components.

In iterative and incremental software development, the implementation process relies onto previous iterations having completed their requirements as further additions and modifications are directly based onto them [12]. This makes the implementation process very sensitive to deviations in the assumed implementation state. As we have predicted this to constitute for the majority of technical debt related observations, the *DebtFlag* mechanism has been designed to integrate into the development environment; as close as possible to the developer and the process emergent to these observations.

Other software development components, emergent to technical debt related observations, are dependent onto the used software development method. The Scrum method's Sprint review [13] is an example of a software development practice the *DebtFlag* mechanism is expected to support. Here developers, who are familiar with the *DebtFlag* mechanism, take part in a process where a software product or a sub-product is assessed against currently active requirements. As deviations are found and documented, the developers give them an implementation level representation in the form of *DebtFlag* elements.

The *DebtFlag* mechanism produces and maintains a TDL according to the captured observations. The TDL can be used to integrate further evaluation and decision approaches from the TDMF. Concurrently the produced TDL provides valuable information to existing software development components. For example the Sprint planning practice of the Scrum method [13] may apply the TDL in defining new backlog items: large TDI entries may require their own backlog items, while the decomposition of new requirements into backlog entries is further defined by reviewing the amount of technical debt indicated by the TDL for this implementation area.

B. Benefits

DebtFlag captures human-made observations. In addition to relevant project stakeholders being fully aware of all active requirements and development conventions, they can provide additional reasoning for their observations. This ensures that information regarding the captured technical debt of a project is both accurate and well defined. Improvements based onto this information should be very effective.

DebtFlag documents the structure of technical debt. Software implementations are complex, hierarchical and interconnected structures. Technical debt that resides in them has similar characteristics. The *DebtFlag* mechanism captures technical debt as Technical Debt Items. TDIs are formed as a set of *DebtFlag* elements for which the *DebtFlag* mechanism

automatically resolves the propagation paths. This structured form allows to track technical debt during continued development but also to apply various assessment approaches to the different levels of the acquired hierarchy.

DebtFlag presents technical debt at the implementation level. By projecting all technical debt observations onto the implementation level, the *DebtFlag* mechanism ensures that development is conducted while aware of technical debt's presence. This allows developers to avoid unintentionally increasing the value of technical debt through dependencies to affected areas or to efficiently decrease its value by resolving technical debt in areas where development is currently conducted.

DebtFlag makes continued use of higher level technical debt management approaches possible. The documentation structure is designed to be able to produce the Technical Debt List for the Technical Debt Management Framework. The extensions, that the *DebtFlag* mechanism introduces, allows to maintain this list automatically during continued development. This makes TDMF reliant management approaches applicable to the development at any given time.

C. Challenges and Limitations

DebtFlag may endorse technical debt accumulation. The documentation tools of the *DebtFlag* mechanism are designed to be as fast and intuitive to use as possible, in order to make capturing technical debt efficient enough to be justifiable. At the same time, the barrier for taking on technical debt is lowered as documenting it consumes less resources than making the optimal implementation. This is an unwanted side effect which is currently remediated by making sure the author of each *DebtFlag* element is documented. Additional measures are devised as case studies provide more information on this possible problem.

DebtFlag places the burden of technical debt management onto the end user. The *DebtFlag* mechanism relies onto the end user for identifying source points for technical debt and for resolving them at a later point in time. This indicates that the burden of technical debt management for the software product is placed onto the end users. This indicates a dependency: the state of technical debt management diminishes directly as a consequence of the end users' inability to identify or to input technical debt information into the *DebtFlag* mechanism as well as the *DebtFlag* mechanism's inability to enforce technical debt governance. In order to overcome the aforementioned problems, we intend to commit case studies to identify problems in user experience and training as well as to further develop the *DebtFlag* mechanism's ability to be cross-compatible with other technical debt identification and assessment tools.

DebtFlag does not protect the information from propagation rule set bias. Both the implementation level representation of technical debt as well as the propagation information generated for the web-interface are dependent onto the used propagation rule set (see Section II-B). While the propagation rule set does not affect the source points for technical debt, they have a large effect onto its modeled propagation and thus onto the management aspects endorsed by the *DebtFlag* mechanism. For this reason, it is important that the propagation rule set used by the *DebtFlag* mechanism is capable of reflecting the actual propagation and technical debt accumulation in the implementation. Acknowledging this, we have started a separate research on more sophisticated technical debt propagation models (see Section V-A).

DebtFlag is heavily dependent onto outside services. Unlike Automatic Static Analysis (ASA) approaches, the *DebtFlag* mechanism requires constant information about the implementation in order to function to its full capacity. While it is possible to recreate the implementation level presentation of technical debt from the database, addition and modification of *DebtFlag* elements is dependent onto having access to the development environment and implementation specific information. As the *DebtFlag* currently supports only the Eclipse IDE and the Java language, this is restrictive.

V. Future Work

We have presented the *DebtFlag* mechanism concept and the tool under development in various discussions. Attendees from both academic and industrial sectors have provided us with valuable initial feedback. Accommodating this, the *DebtFlag* tool is expected to reach its first major version during the first quarter of 2013. This will enable us to commit case studies to improve, validate and extend both the mechanism and the tool.

A. Mechanism Improvement and Validation

The current schedule will allow us to start conducting case studies with the *DebtFlag* tool during the second quarter of 2013. Here, we will first concentrate on improving the mechanism by finding solutions to the challenges and limitations presented in Section IV-C. As our department currently plays host to a variety of research where large scale software development is carried out using iterative and incremental development approaches, the first case study will be conducted in-house in order to retain the controlled environment.

This case study will be started with a thorough mapping of current product state as well as used implementation techniques and practices. This is followed by introducing the *DebtFlag* tool to the project combined with appropriate training and instruction. During continued development, we will respond to developer feedback in order to discover deficiencies in training and to enhance the user experience of the *DebtFlag* tool. Simultaneously, we will be upholding a manual identification and assessment process to gather information on technical debt and its propagation. During control periods we will be examining the differences in upholding the product's TDL with the *DebtFlag* tool and the manual process in order to discover differences between the two approaches. According to these results we will improve the *DebtFlag* mechanism.

We do not expect the aforementioned case study to solve the discussed problem of propagation modeling (see Section IV-C. Anticipating this, we have started a separate research

to overcome this matter. Albeit in early-stages, our research on applying link structure algorithms, especially the PageRank algorithm [14], has provided us with promising results when used to value and indicate most crucial implementation elements for the accumulation of technical debt.

As we have intended this tool as a productivity enhancement for industrial settings, we intend to further improve and validate the *DebtFlag* mechanism in such environments. For this, we have planned and discussed a rather extensive series of case-studies to be committed with a department of a large telecommunication company. To be launched later this year, these case studies will have access to a multitude of data spanning over finished and ongoing iterative and incremental software projects. For finished products we use proven technical debt identification and assessment tools in order to simulate the life-span of technical debt. The results of various propagation models are compared against this in order to provide the *DebtFlag* mechanism with a more sophisticated propagation rule library.

For ongoing projects, we will work closely with the aforementioned party and local software development companies, in order to discover the current state of technical debt and its management for each studied software project. We will then use a refined version of our academic case-study to introduce the *DebtFlag* tool and the manual technical debt management process for these projects. During continued development, we expect to discover ways to further support the projects' technical debt management through enhancements to the *DebtFlag* mechanism. As the studied software projects will form an extensive representation of possible software development approaches, we expect that the results of these case studies will provide the *DebtFlag* mechanism and tool with adequate validation.

In committing the later case studies, we will be covering projects working on legacy software. As the *DebtFlag* mechanism is designed to capture the deviation between the current product state and its requirements, we foresee it being used to produce a mapping between legacy software components and a new requirement set. In such settings the TDL can serve as input for the modernization plan. We expect these case studies to yield additional validation for the mechanism in the form of increased legacy software development efficiency.

B. Extending the Range of Supported Techniques

After accommodating the improvements discovered by our first case study, we will extend the range of supported techniques in order to prepare the mechanism for industrial use. The plans currently encompass extending the current Eclipse plug-in to support Javadoc through the Eclipse JDT and the Python language through Eclipse pyDev, while replicating the plug-in to the Visual Studio environment in order to support the C# language.

In addition to covering a range of implementation and documentation techniques, we will be working on making the *DebtFlag* mechanism cross-compatible with other technical debt identification and assessment tools. By working together with mature technical debt identification and assessment tools (e.g. SQALE [15]) we expect to increase the accuracy and range of produced information, thus making technical debt management more robust with the *DebtFlag* mechanism (see Section IV-C).

REFERENCES

[1] W. Cunningham, "The WyCash portfolio management system," in *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (OOPSLA)*, vol. 18, no. 22, 1992, pp. 29–30.

[2] 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 *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE, 2012, pp. 45–48.

[3] E. Dustin, J. Rashka, and J. Paul, *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

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

[5] M. Friedman and J. Voas, *Software assessment: reliability, safety, testability*. John Wiley & Sons, Inc., 1995.

[6] J. Kupsch and B. Miller, "Manual vs. automated vulnerability assessment: A case study," in *First International Workshop on Managing Insider Security Threats (MIST)*, 2009, pp. 83–97.

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

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

[9] J. McGregor, J. Monteith, and J. Zhang, "Technical debt aggregation in ecosystems," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE, 2012, pp. 27–30.

[10] Eclipse Foundation, "Eclipse integrated development environment," *URL: http://www.eclipse.org/*.

[11] M. Grönroos *et al.*, *Book of Vaadin*. Vaadin Limited, 2011.

[12] T. Gilb and G. Weinberg, *Software metrics*. Winthrop Publishers, 1977, vol. 51.

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

[14] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[15] J. Letouzey, "The SQALE method for evaluating technical debt," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE, 2012, pp. 31–36.

II

Paper II

Software Implementation Knowledge Management with Technical Debt and Network Analysis

Holvitie, Johannes (2014). In *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on*, pages 1–6. IEEE

Software Implementation Knowledge Management with Technical Debt and Network Analysis

Johannes Holvitie
TUCS - Turku Centre for Computer Science &
Department of Information Technology, University of Turku
Turku, Finland
jjholv@utu.fi

***Abstract*—Modern, fast-phased, iterative and incremental software development constantly struggles with limited resources and a plethora of frequently changing requirements. This environment often requires the development projects to intentionally — for example through implementing quick-and-dirty — or unintentionally — for example through misinterpretation of requirements — deviate from the optimal product state. While most of the deviation is caught through practices like customer reviews, the remainder stays hidden in the product. The undocumented remainder is difficult to remove, it expands uncontrollably and it negatively affects development as deviations are unexpectedly encountered and overcome. The term technical debt describes this process of accumulating hidden work. Management of technical debt can be expected to be a major factor in software development efficiency and sustainability and as such it should be an integral part of the software implementation's knowledge management. In addition to being difficult to capture, the continuous evolution of the implementation makes maintenance of gained information a challenge. This paper discusses applying technical debt management for software implementations including the entry points for knowledge discovery, network analysis for overcoming the maintenance challenges as well as the pursued outcomes.**

Keywords*—technical debt management, network analysis, program visualization, refactorization.

I. Introduction and Problem Statement

Modern software development methods deal with increasing complexity and frequently changing requirements by decreasing increment size and shortening iteration times. These, often Agile or Lean, methods try to imply control over the product in requiring that all releases meet the definition of done. For the client this definition encompasses the product's perceivable capability to fulfill set requirements. The developer's definition is a super-set of this in also capturing how the requirements are implemented. This dualism of definitions allows iterations to deliver increments which meet the former but not the latter, that is they are perceived done while actually being incomplete.

This is a very common phenomenon in software development resulting from developer actions that are either intentional, like relaxing quality requirements when pressed for time, or unintentional, like making uneducated design decisions. Since the opposing party, the client, does not perceive these inconsistencies it can not pursue their completion. Rather, the developing organization is implicitly assumed to manage them. But, when faced with the choice of allocating resources to implement perceivable requirements or fixing unperceivable inconsistencies, the choice of the former can be expected — especially when no information supporting the latter is readily available. These decisions and the discussed phenomenon respectively accumulate new and increase existing technical debt.

Technical debt considers the deviation between the current and optimal product state and it is problematic due to a number of reasons. Firstly, as its emergence is the end result of a rather obfuscated process, it rarely gets documented. Low visibility makes management difficult and it hides problem severity. Technical debt has the ability to accumulate super-linearly as software solutions are built by depending onto earlier, unsuccessfully implemented, components. And even if technical debt gets indicated the evolving implementation promptly degrades this information.

Due to the multiple issues exposed by technical debt, software development projects require mechanisms that are able to efficiently capture, track and govern it. The mechanisms need to accomplish these while retaining development characteristics like agility. The mechanisms should also accommodate the fact that some instances of technical debt exist due to informed decisions.

This paper discusses research to overcome these challenges and it is structured as follows. Section II discusses motivation for this research in more detail while Section III derives the main hypothesis and work objectives. In Section IV work related to technical debt and its management is presented with accompanying terminology. Section V proceeds to layout the research plan to accomplish set objectives while Section VI discusses the expected outcomes. Section VII concludes this paper.

II. Motivation

There exist mechanisms, like bug reports and unit tests, that are able to capture the deviation between the current and optimal software implementation states, but none of these consider the dynamic aspects of software development. The Agile Manifesto's [1] 10^{th} principle states that "*software development should pursue simplicity - the art of maximizing amount of work not done*". In respect to this, it is not ideal to immediately fix every encountered non-conformality, since this may amount to a lot of unnecessary work, but at the same time total ignorance can not be exercised as some of the non-conformalities may end up causing much more work than the initial fix would have required.

Technical debt is about acknowledging these dynamic aspects in order to optimally manage the singular deviations. Technical debt management pursues the formation of technical

978-1-4799-2393-9/14/$31.00 ©2014 IEEE

debt instances [2], units of deviation that share the same context and are thus governable by a single person or team. This enables documenting the impact probability for each instance, capturing the chances that it causes additional work during continued development. Further, the instances allow estimating accumulated impact for them as they become more coupled to the system. The estimates can then be fed back to project decision making in order to make technical debt — or optimized deviation — management a concrete part of it.

As technical debt management does not concern with different types of non-conformalities it is possible to superimpose it on top of existing mechanisms. Additionally, this allows considering contexts composed from elements at different implementation abstraction levels, in different project artifacts or purely subjective observations as manageable instances.

While technical debt exposes these dynamic characteristics, it does not expose a model or a mechanism for maintaining the captured information. Thus, the greatest challenge in introducing technical debt management to software development lies in deriving feasible maintenance approaches. In technical debt management research (see Section IV), one discussed approach to this considers manual updating a possibility. The research presented in this paper argues that software development, especially for the implementation artifact, is carried out based on a number of underlying models for which network analysis can reveal static update models. A hybrid approach of initial manual input and continued updates through a static model could retain a high level of accuracy while supporting automated maintenance.

III. Hypothesis and Objectives

Research work discussed in this paper facilitates exploring a general hypothesis through a more specific one. As technical debt management pursues optimized governance of non-conformalities in software development, a general hypothesis for this is stated as *technical debt management is a factor in software development efficiency and sustainability*. As the non-conformalities are exposed, made into explicit instances, and input into the decision making processes, the uncertainty about required additional work decreases. This is expected to result in more streamlined development and thus increased development efficiency. Sustainability is increased based on the same grounds: as non-conformalities and their impact on to development are made explicit, the developing organization is provided with means to perceive and control the problem before it becomes unbearable.

The more specific hypothesis, or task, studied here facilitates the general one. It concerns the maintenance challenges in the software implementation context (see Section II) in stating that *software implementation technical debt can be captured and maintained*. Captured via tools that are present when notions about technical debt are made and maintained via a model that takes the subjective notions and incrementally updates them. These increments require a static reference, a model, that dictates granular updates for the structured notions based on observed evolutionary steps in the software implementation.

Three consecutive work objectives facilitate studying the specific hypothesis. The first step concerns *building a tool capable of capturing technical debt instances*, the notions discussed earlier. The second step *builds the static model* by applying network analysis to programming theory, the user inputs and captured technical debt propagation characteristics. The third step combines the tool with the model, inputs the model with structured notions about technical debt and observed evolutionary steps in the implementation *to derive information updates*. The specific hypothesis is studied by determining if the information captured and maintained in the third objective is successful in serving technical debt management approaches. Quantifying enhancements to technical debt management can be seen to further efforts to overcome the general hypothesis.

IV. Related Work and Terminology

In meeting the work objectives discussed in Section III, a number of research contexts are consulted. Building the tool for the first objective, the documentation structure needs to accommodate requirements set by technical debt management. The update model derived in the second objective bases on identifying and separating technical debt instances from software implementations and applying network analysis to capture shared and unique attributes. In the following sections essential research and terminology are introduced for these areas.

A. Technical Debt

Technical debt is a term that was first coined by Ward Cunningham in his technical report to OOPSLA'92 [3]. Updated definitions have followed this for example in Seaman et al. [4] as well as in Brown et al. [5]. A consensus between these definitions is that technical debt is based on a principal on top of which interest is paid — similarly to its financial counterpart. The principal captures the original, deliberately or indeliberately, incurred non-conformality. The size of the principal is equal to the work required to provide an optimal solution for it. The interest is relative to the amount of adaptation the surrounding system has committed to when the principal has become more coupled to it. Seaman et al. [2] expand the definition of interest by stating that interest is an impact value coupled with impact probability. This probability communicates about the chances of development work being continued in the interest's implementation area. For example, if no further development is carried out in this area, the probability of needing to pay the interest is zero.

Technical Debt Management Framework (TDMF) introduced in Guo et al. [6] is a software development method independent approach for integrating technical debt management into project decision making. The two first steps of this three part approach build a Technical Debt List (TDL) which captures the technical debt instances for a software project. The instances are captured as Technical Debt Items (TDI) following a documentation structure dictated by the TDMF. Most notably, this structure requires that for each TDI, explicit impact size and interest probability are estimated. As the TMDF enables integration of and information interchange between other management approaches, the tool pursued for the first work objective will adhere to it.

McGregor et al. [7] were the first to discuss about technical debt's propagation in software implementations. In their work

they hypothesize about two concurrent mechanisms. The first states that "*technical debt for a newly created asset is the sum of the technical debt incurred by the decisions during development of the asset and some amount based on the quality of the assets integrated into its implementation*". During introduction of this mechanism, they communicate about technical debt's ability to diminish as a result to increases in interface layers. The second mechanism describes that "*the technical debt of an asset is not directly incurred by integrating an asset in object code form, but there is an indirect effect on the user of the asset*".

B. Capturing Information from Software Implementations

Software implementations are structures in which components rely onto others in order to fulfill their functionality. Capturing these structures as sparse matrices either dynamically, from the program execution trace, or statically, from data mining the source code, allows the application of network analysis approaches to further analyse and understand them.

Link structure algorithms produce either global or query-specific importance rankings for matrix elements. The PageRank algorithm by Page and Brin [8] has received most of the attention regarding producing global rankings for implementation components. In amongst others [9]–[11] have introduced their own adaptations of the algorithm for use in analysing software implementations. They have acknowledged the possibilities in using this approach for example in valuing components for software impact analysis as well as relating implementation elements for feature location. An important notion from these works is that even the slightest changes in building the matrix result in large fluctuations in the received rankings.

Program visualization is, in some regards, a more humane approach to link structure analysis. Interesting observations can be quickly made even for a complex context if it is possible to produce a visualization for it and especially to highlight structures of interest within it. Caserta and Zendra present a program visualization classification in [12]. In this they state that *graph* as an approach is a forerunner for architecture level visualizations that focus on highlighting relationships. Noack and Lewerentzes discuss a requirements space for visualizing software implementations with graphs [13]. They formalize it as the three degrees of clustering, hierarchicalness and distortion. For approaches interested in exploring directly visible structural relations, all previous degrees should be low.

V. Approach and Challenges

The author's study focuses on capturing and maintaining technical debt information for the software implementation in order to facilitate integration of technical debt management for it. Section III described the three work objectives that were derived to overcome this: creating a tool capable of capturing technical debt instances, a static model to automate maintenance for them and integration of the model with the tool to produce a fully functioning management suite. At the current state, a solution exists for the first objective while research is underway to facilitate the second one. This chapter walks through the three objectives describing the chosen approach, existing work and foreseeable challenges for each.

A. First Objective - Technical Debt Management Tool

The first objective required a tool that was readily available when notions about technical debt were made in the implementation. To overcome this the author co-designed a two-partite tool called *DebtFlag* which is presented in [14]. The first part of this tool (see Fig. 1) is a plug-in for the Eclipse integrated development environment (IDE). The plug-in allows developers to create TDIs by interacting directly with edited implementation elements. The TDIs are bound to revisions and thus are synchronized through the version control system. In addition to ensuring that intrinsic information about the TDI's impact and propagation characteristics is recorded, the plug-in also provides a representation for them. The plug-in introduces an update mechanism that uses the static update model from Section V-B to calculate where given TDIs propagate. Affected implementation elements are highlighted in colors according to given rules. The highlights in both the editor view as well as the content-assist (see Fig. 2) make it virtually impossible to carry on implementation without knowledge about technical debt's presence. This mechanism is one of the first concrete tool solutions to controlling unwanted technical debt propagation.

The second part of this tool is an overview web-application (see Fig. 2). The plug-ins are to communicate TDIs to a

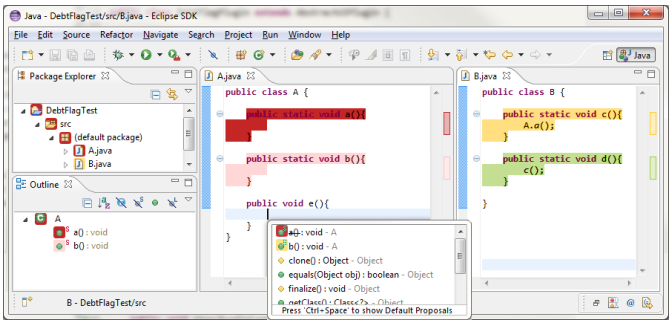


Fig. 1. The *DebtFlag* plug-in managing code highlighting and content-assist cues in the Eclipse IDE

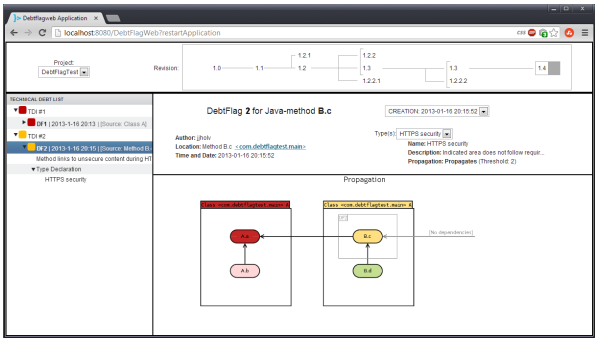


Fig. 2. The *DebtFlag* tool's web interface

database from which the web-application builds project specific TDLs. Basic functionalities, like modifying details for TDIs and observing their history as a function of revision commits, are available through the web-application. The TDLs can be used as is for inter-organization communication but they are also intended to provide an interface for integrating other technical debt management approaches to the projects.

The most formidable challenges regarding the tools use affect application of the captured information. Firstly, maximized efficiency and intuitiveness for the tool's usage tries to encourage developers to capture all technical debt they perceive. Unfortunately, this also presents a way to measure performance for the authoring persons. This would inevitably discourage further use of the tool and negatively affect the approach's viability. The presenting paper [14] discusses certain levels of information hiding as a remedy for this. The other challenge is closely related to this: if authors for TDI entries are hidden and producing them is easy, there is a danger that for TDIs for which the correct action would have been to directly repair them, instead a TDI entry is made. Expediting repairs for small entries through valuing them higher in the static model is discussed as a possible answer to this.

B. Second Objective - Static Update Model

The second work objective concentrates on building a static update model for the *DebtFlag* tool introduced in the previous section. The tool queries the model with indicated locations and expects a (fuzzy) set of elements as output — indicating technical debt expansion for them. This objective is a derivative from the technical debt integration requirement by Seaman et al. [4] which states that software project decision processes need to be accompanied with usable and current information regarding the project's technical debt. Introducing automated maintenance for manual observations allows efficient extension of their applicability thus retaining more information for decision making.

The author has partaken in a number of studies to facilitate exposing the static models. In the first such study we examined the role of dependency propagation in the accumulation of technical debt by conducting a manual retrospective analysis for a large refactorization project [15]. Based on captured relations at the class level, the following observations were made. First, the number of incoming dependencies to an implementation element correlated with the number of propagation paths for technical debt. Second, dependency propagation was the main driver for technical debt accumulation. This was evident from that for a majority of chronological modifications a direct dependency relation was observed.

Third, technical debt diminished due to propagation. This was a result from measuring propagation depths to be smaller than what component dependencies would have allowed for. Fourth, the component's role was a good indicator of the propagation's shape. This was apparent from observing that for technical debt affecting data models the reparations expanded in the system in a shape that was wide but shallow, while for reparations targeting direct functionalities the shape was more focused but deeper reaching. In our yet unpublished journal extension *Examining Technical Debt Accumulation in Software Implementations*, we studied the four observations for another independent data set at a lower, class-member, level. Finding that data supported the observations here as well was perceived as an indicator for technical debt's universal propagation capabilities. As such, the initial static model is required to accommodate at least the four observed characteristics.

To facilitate studying technical debt and its propagation characteristics for large implementations and non-conformality counts, a program visualization approach was designed. We demonstrate this in a yet unpublished study *Illustrating Software Modifiability - Capturing Cohesion and Coupling in a Force-Optimized Graph*. The approach has three consecutive steps to forming the visualization. The first step traverses a source implementation and captures all program elements that are capable of forming direct dependencies in it. The second step forms a graph by presenting the implementation elements as nodes and capturing the direction and frequency of dependency invocation between them as the graph's directed and weighted edges. The third step lays out the graph through force-minimization. In this, the directed weighted edges represent forces and finding a global energy minima for such a system emphasizes those structures that contribute towards it.

Fig. 3 demonstrates applying the visualization approach to the source code of the Eclipse IDE's Debug component. Here,

nodes represent Java interface members and edges capture their invocations. The gray graph is the component's part from the Eclipse's implementation force-minimally laid out. The distinct hub in the upper part is the component's more independent, cohesive and less coupled, event system. The highlighted part in the bottom is the component's bug #148965. The red and green lines indicate dependencies that are outbound and inbound respectively to elements declared for the bug. The highlighting mechanism allows us to quickly inspect how non-conformalities propagate. In this case, the green lines indicate that the root cause for this bug maybe coming from outside the Debug component's implementation.

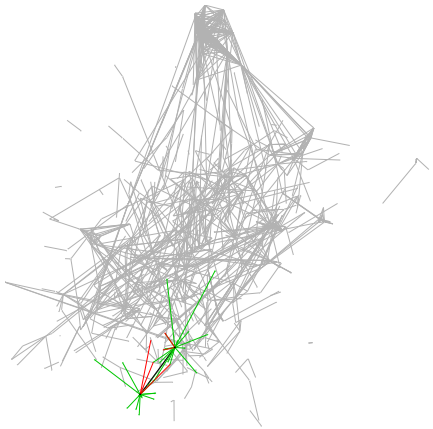


Fig. 3. Presents Eclipse's bug no. 148965 highlighted against its host Debug project graph

Variations of the static update model are currently being trialed to effectively account for the four observations that were described earlier. The aforementioned visualization approach allows to efficiently handle large implementation data sets and to distinguish special characteristics from them. The network analysis approaches discussed in Section IV-B are used on the produced graphs to rank their elements and produce correlation coefficients. Distinguished special characteristics are then extracted as components for the static model using basic functional regression and related machine learning approaches.

There are two perceivable challenges to this work objective. The first is the applicability of the model. A model that covers multiple programming languages will not take into account intrinsic details for each one. This is a trade-off between applicability and accuracy. Additional research needs to balance this as it is important to produce approximations with a minimum of false-positives since these are directly used to produce the technical debt representation. Secondly, the propagation model training sets as well as its inputs may contain speculation. For instance the bug in Fig. 3. The location initially indicated for the bug can be speculative and the true location is only known when it has been fixed and verified as such. This must be taken into account.

C. Third Objective - Maintaining Captured Information

The third and last work objective will take the static update models from the second objective and inputs them to the *DebtFlag* tool from the first objective. As stated, the *DebtFlag* tool exposes an abstract update descriptor and is thus able to accommodate all static update models that inhere to it. Field testing of different models is thus conducted through the *DebtFlag* tool which puts further emphasis on its implementation and user experience quality.

As the third objective corresponds to examining the specific hypothesis introduced in Section III, it will be overcome in two stages. The first stage will take the most promising models, inputs them to the *DebtFlag* and approaches a small number of organizations from varying development contexts. This stage will provide both the tool and the used models with enhancements. Having accommodated them, the second stage will be conducted as a quasi-experiment where the tool is introduced, with the chosen models, to controlled development organizations. For these organizations, their implicit and explicit technical debt management approaches need to be identified so as to be able to observe how the tool affects them.

This objective decompiles into proving the specific hypothesis, which stated that software implementation technical debt can be captured and maintained. Measuring this as project work-in-progress capability increments as well as improvements in estimate confidence comes very close to examining the general hypothesis for technical debt management. The main challenges here are those shared by all controlled experiments. What is expected to be especially challenging is the derivation of valid measuring points and relation of gained results with control groups.

VI. Contribution

The contributions of the presented research come from accomplishing the three work objectives discussed in the previous chapter. As it currently stands, the author has completed the first work objective and is working towards completing the second one. Completion of both the first and second objective is required to pursue the third objective, which constitutes majority of the presented researches' contribution. However, several advances can already be seen and they are discussed in the following.

The *DebtFlag* tool from the first objective is the first in its kind to explicitly pursue technical debt management as part of software development. Close integration with development tools allows the tool to provide an implementation level representation for captured technical debt, which makes technical debt unaware development impossible while also enabling micro-management for singular technical debt instances. This should result in notable efficiency improvements when taking into account technical debt's super-linear accumulation speed.

Capability to explicitly manage implementation components prone to technical debt has lead to further studies in which the *DebtFlag* tool is applied to overcome issues in legacy software development. The ability to produce a "legacy interface" for the project allows to efficiently restrict incoming dependencies to these parts while developers work towards adapting the legacy parts to the rest of the project.

Research to expose properties of technical debt propagation in the second objective allows capturing them in static update models. Capability to automatically update manually made observations during continued development allows extending their applicability. Such a mechanism is required to pursue optimizing defect governance. Additionally, exposing and reporting on found propagation characteristics hopefully leads to further research on the area, possibly emergent to complementary update models.

Finally, integration of the static models with the *DebtFlag* tool in the third objective is expected to produce a fully functioning technical debt management suite. Capturing notions in a way that adheres to the TDMF documentation requirements should produce a medium through which developers can easily communicate about technical debt and its resource requirements to the management. Underlying update model ensures that all information is current and thus applicable in decision making. Adherence to the TDMF's documentation policies also allows this development method independent suite to act as an interface to integrate further technical debt management policies into these projects.

VII. Conclusion

Technical debt captures the uncertainty for a software project and communicates about the effects it causes. On the highest level these can be seen to include declining development efficiency and sustainability. Uncertainty in development calls for reserving more resources in order to overcome possibly encountered issues. This leads to a less streamlined and less efficient process. Having fewer resources available reduces development sustainability. The project becomes more rigid and less capable of accommodating quickly changing requirements. That is, technical debt hides the project's true state and leads to decisions being made disconnected from it: unaware of the project's actual capabilities to overcome requirements and unforeseen risks while ignoring the optimal moments for reductive maintenance.

Research proposed in this paper facilitates introducing technical debt management for software implementations. As they usually constitute majority of the projects' accumulated value, the effects of technical debt and its management are felt the strongest here. Ability to capture and maintain information about software implementations' technical debt does not only allow the introduction of further management approaches but also the introduction of this information to existing approaches so as to make them sensitive to technical debt as well.

Three consecutive work objectives were discussed to achieve this. The first objective called for a tool that allowed subjective notions about technical debt to be captured in a structured manner. The *DebtFlag* tool was introduced as a solution to this. In addition to supporting the TDMF, the *DebtFlag* introduced a novel micro-management approach for technical debt. The second objective finds static models to be used in updating the captured notions. Research towards this is currently underway, having already distinguished a number of general propagation characteristics for technical debt while continued research tries to identify and model the unique characteristics of specific implementation techniques. The third and final objective will then combine the two former ones in order to produce a fully functional technical debt management suite to overcome the matters discussed in the previous paragraph.

The author expects the pursued approach to result in a number of enhancements for software implementation technical debt management with practical applications. Even small advancements should be considered as the iterative and incremental properties of current software development methods multiply the effect in the host project. At the same time, these methods correspond to the research's greatest challenge as technical debt management needs to integrate to them without suppressing their unique characteristics.

References

[1] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.*, "Manifesto for agile software development," 2001.

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

[3] W. Cunningham, "The WyCash portfolio management system," in *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (OOPSLA)*, vol. 18, no. 22, 1992, pp. 29–30.

[4] 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 *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE, 2012, pp. 45–48.

[5] 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.

[6] 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 *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 528–531.

[7] J. McGregor, J. Monteith, and J. Zhang, "Technical debt aggregation in ecosystems," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE, 2012, pp. 27–30.

[8] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[9] B. Turhan, G. Kocak, and A. Bener, "Software defect prediction using call graph based ranking (cgbr) framework," in *Software Engineering and Advanced Applications, 2008. SEAA'08. 34th Euromicro Conference*. IEEE, 2008, pp. 191–198.

[10] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *Software Engineering, IEEE Transactions on*, vol. 31, no. 3, pp. 213–225, 2005.

[11] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 419–429.

[12] P. Caserta and O. Zendra, "Visualization of the static aspects of software: a survey," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 7, pp. 913–933, 2011.

[13] A. Noack and C. Lewerentz, "A space of layout styles for hierarchical graph models of software systems," in *Proceedings of the 2005 ACM symposium on Software visualization*. ACM, 2005, pp. 155–164.

[14] J. Holvitie and V. Leppänen, "DebtFlag: Technical Debt Management with a Development Environment Integrated Tool," in *Managing Technical Debt (MTD), 2013 Fourth International Workshop on*. IEEE, 2013.

[15] J. Holvitie, M.-J. Laakso, T. Rajala, E. Kaila, and V. Leppänen, "The role of dependency propagation in the accumulation of technical debt for software implementations," in *13th Symposium on Programming Languages and Software Tools*, k. Kiss, Ed. University of Szeged, 2013, p. 6175.

Paper III

Examining Technical Debt Accumulation in Software Implementations

Holvitie, Johannes and Leppänen, Ville (2015). *International Journal of Software Engineering and Its Applications*, 9(6):109–124

International Journal of Software Engineering and Its Applications
Vol. 9, No. 6 (2015), pp. 109-124
http://dx.doi.org/10.14257/ijseia.2015.9.6.12

Examining Technical Debt Accumulation in Software Implementations

Johannes Holvitie and Ville Leppänen

Turku Centre for Computer Science (TUCS) &
Dept. of Information Technology, University of Turku
FI-20520, Turku, Finland
{jjholv, ville.leppanen}@utu.fi

Abstract

Technical debt management requires means to identify, track, and resolve technical debt in the various software project artifacts. There are several approaches for identifying technical debt from the software implementation, but they all have their shortcomings in maintaining this information. Readily available information is a cornerstone of successful technical debt management integration. As such, this paper presents a two-partite case study that explores the role of dependency propagation in the accumulation of technical debt for software implementations. The first part, targeting a refactorization project in the ViLLE education platform, identifies a clear relation between the propagation and the accumulation in addition to making notions about special propagation characteristics. The second part considers bug reports for the Eclipse IDE, and, thus, provides further verification in observing a similar relationship at a lower implementation abstraction level for this independent data set. We conclude that formalization of this relation should lead to solutions for the technical debt information maintenance problem. As such, we use the case study herein to further improve the propagation model applied in our DebtFlag technical debt management tool.

Keywords: *technical debt, technical debt propagation modeling, software implementation assessment, refactoring*

1. Introduction

Technical debt is a metaphor that describes how various trade-offs in design decisions affect the future development of the software project. Trade-offs are made between development driving aspects —for example meeting a delivery date by relaxing some quality requirements— and they incur the project's technical debt while providing the organization with a short-term gain. Similarly to its financial counterpart, technical debt can also accumulate interest over a principal until it has been paid back in full —for example through reuse in software implementations. Inability to manage the projects technical debt results to increased interest payments in the form of additional resources being consumed when implementing new requirements, and ultimately to exceeding development resources and premature ending for the project [5].

Technical debt management is a software development component and an actively researched area of software engineering [15]. It is interested in providing projects with means to identify, track, and payback technical debt in order to provide similar control to technical debt as there exists for other project components. There are various software project artifacts, such as process, testing, architecture, implementation, and documentation, that are prone to the aforementioned decisions and thus to hosting technical debt. As these fields differ from each other to a large degree, techniques for managing technical debt are separate for each of them.

ISSN: 1738-9984 IJSEIA

For the software implementation artifact, we can divide the technical debt identification techniques into automated [13] and manual approaches [19]. The challenge here is that the information produced by either of these approaches is only applicable to the assessed implementation version: automated approaches can produce results for all implementation versions, but they only highlight modules that violate the input static model. This leaves out information regarding module relations and links to previous implementation versions. Manual approaches on the other hand do provide some information regarding the history of a certain technical debt occurrence, but update frequencies to this information make these approaches only capable of tracking and managing technical debt on higher levels. These observations have led us to conclude that if the relation between software implementation updates and increases in technical debt could be made explicit, we could automatically extend the applicability of manually produced and accurate technical debt information to future versions. This would greatly increase the efficiency of technical debt knowledge management for software implementations.

In this paper, we present a two-partite case study that explores the aforementioned possibility. Building on the assumption that dependency propagation, the main construction mechanism for software implementations, drives technical debt accumulation in this area, we explore the relationship by deriving two objectives for this case study: 1) to identify technical debt and its structure in the studied implementations followed by 2) establishing the role of dependency propagation in the formation of these structures. These objectives are studied for two data sets. The first expands on the results of a separate refactorization project, and considers propagation at the class level. The second verifies if similar conclusions can be made for reparations made on a lower, class-member, and level.

This article extends our earlier publication titled *The Role of Dependency Propagation in the Accumulation of Technical Debt for Software Implementations* [11]. Sections 3 through 5 of this paper contain a revised and expanded version of the case study described in the aforementioned publication. Sections that follow this provide a detailed description for the new, verifying, and larger case study conducted for the class-member level. The full paper structure is as follows. Section 2 over goes related work for technical debt, its management, and propagation in software implementations. Followed by Section 3 which describes the research problem and the study's structure while introducing the target system for the first part. Results for this are discussed in Section 4 while conclusion and validity issues for it are discussed in Section 5. The target system and study execution for the second part are then described in Section 6, while conclusions are made and contrasted against the former part in Section 7.

The presented study is part of a research into establishing if a tool-assisted approach can be introduced for software projects in order to efficiently identify, track, and resolve technical debt in implementations while they are under development. The results of this case study will be used to further develop the DebtFlag technical debt management tool [9] (see Figure 1) and its propagation model for technical debt. The DebtFlag-tool is a plug-in for the Eclipse IDE and it implements the DebtFlag-mechanism, for maintaining technical debt notions (see [9] for a detailed description). The tool is used to identify technical debt instances from the implementation and to merge them into entities allowing management at both the implementation and project levels.

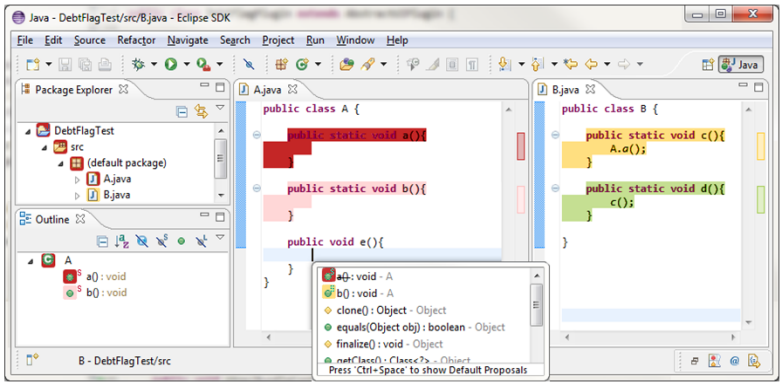


Figure 1. Debtflag Code Highlighting and Content-Assist Cues in the Eclipse Ide

2. Technical Debt

The term technical debt was first introduced by Ward Cunningham in his technical report to OOPSLA'92 [5]. Complementary definitions have been provided, in amongst others, in the works of Brown, *et al.,* [4] and Seaman, *et al.,* [20]. A general consensus between these definitions is that technical debt bases on a principal on top of which some interest is paid. The principal corresponds to the size and amount of unfinished tasks that emerge as design decisions make trade-offs between development driving aspects. Principal is paid back by correctly finishing these tasks. Interest is increased by making more solutions depend upon areas where there are unfinished tasks. When creating these dependent solutions, if additional work is required due to inoptimality in these areas, this constitutes as paying interest. Seaman, *et al.,* formalize this further by defining interest as an occurrence probability coupled with a value [19]. The occurrence probability takes into account that not all technical debt affects the project: for example if a part of the software implementation is never re-used, the probability of this part hindering further implementation updates is zero.

It must be noted that the young age of the field still shows in the previous definitions. In [4], a very detailed and structured definition is given for the technical debt concept, but only on the abstract level. This is problematic, especially in our case, where we must apply the definition in the extraction process. Further, as both definitions in [20] and [19] extend the concept to technical debt management, the problem of repay probability adjustment is encountered. In both cases, the problem is not fully overcome, as the probability of technical debt becoming payable is mostly affected by the unforeseeable future development environment. Again, surveys similar to the ones presented herein, must, in almost all cases, limit to retrospective analysis of technical debt. Since, in these cases, the technical debt has become repayable and is also repaid.

Management of technical debt can be either implicit —like in many agile software practices, where reviews are made during and in between iterations to ensure that the sub-products meet the organizations definition of done— or explicit —like employing a variation of the Technical Debt Management Framework [19, 8]. In either case, the success of technical debt management is largely, if not solely, dependent onto the availability of current technical debt information [20].

2.1 Technical Debt in Software Implementations

Following the definition of technical debt we can see that for software implementations the unfinished tasks are components that, in their current state, are unable to fulfill their requirements. The size of these tasks corresponds to how difficult it is to finish each component, and together they form the principal of the software implementation's technical debt. Similarly, we can see how the interest of technical debt forms in software implementations: dependency onto unfinished components indicates that the dependent may have had to accommodate this in some manner. This accommodation accounts as increased interest for the depended upon component's principal, and if the amount of work required to implement the dependent is increased then this corresponds to paying interest.

To clarify, in the previous paragraph, a software implementation component refers to an entity that is defined by the used programming paradigm and technique, and is capable of forming dependencies. Both of the target systems in this case study are implemented using the Java programming language. Here, like in many object oriented languages, direct references and inheritances create dependencies to public interfaces formed out of variables and methods [2].

In order to maintain the technical debt information produced either by means of automatic or manual identification there needs to exist a model describing how technical debt propagates in the software implementation. Theories for this are discussed in Section 2.2. Additionally, certain implementation technique and paradigm specific characteristics need to be taken into account when identifying possible propagation routes for technical debt. Especially interfaces which can hide portions of technical debt or decouple dependents from refactorizations.

Software implementation's technical debt is paid back by repairing the product. For reparation of non-functional requirements, we follow Fowler, *et al.*, [7] definition of refactoring: *"changes made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"*. For functional requirements, the same procedure applies, but the observable behavior should meet expected functionality. In the following, this is used as basis for identifying software components affected by technical debt.

2.2 Related Work on Technical Debt Propagation

McGregor, *et al*., [14] hypothesize that technical debt has the ability to aggregate within elements of the software implementation and provide two concurrent mechanisms for it. The first one stating that *"technical debt for a newly created asset is the sum of the technical debt incurred by the decisions during development of the asset and some amount based on the quality of the assets integrated into its implementation"*. In respect to this, they note that technical debt may diminish as a result of increased implementation layer nesting. The second mechanism dictates that *"the technical debt of an asset is not directly incurred by integrating an asset in object code form, but there is an indirect effect on the user of the asset"*. For a software implementation, this can mean for example that the implementation of a new element does not necessarily increase the technical debt quota, but deficiencies in the documentation still result into more consumed resources.

Research is scarce in relating technical debt accumulation with the mechanics of software dependency propagation. We refer to research on software evolution and change impact analysis to gain insight into dependency propagation and its characteristics. Avellis [1] discusses the implementation of a change impact function, and notes that, for domain-specific areas, the information encoded into the domain models can be used to parameterize the change propagation rules while monitoring the ripple-effect of a change requires deep knowledge about the modification's implications. It is also concluded that

the use of more specialized information in the definition of the propagation paths results into a more specific and accurate impact set. This, however, leaves room for ambiguity. Notably, it is not fully clear what aspects of the domain model should constitute in the change propagation parameterization, are there certain subjective bias concerns regarding the discussed deep knowledge, and can introduction of specialized information negatively affect recall.

In Bianchi, *et al.*, [3], the authors note that the number of outgoing dependencies from a component is related to the number of paths through which the effects of a change may propagate. While a strong correlation can be seen for this mechanism, others can also co-exist. When developing propagation models, we must be wary of them. Robillard [18] presents an algorithm for providing an interest ranking for directly dependent change candidates. The ranking of elements is based onto *specificity* and *reinforcement*, where the former rules that *"structural neighbors that have few structural dependencies are more likely to be interesting because their relation to an element of interest is more unique"* and the latter that *"structural neighbors that are part of a cluster that contains many elements already in the set of interest are more likely to be interesting because …[they] probably share some structural property that associates them to the code of interest"*. In respect to this, it should be noted that while certain paradigms, like object-orientation, give evidence for this; other, possibly newer, paradigms and techniques require additional investigation.

3. First Case - Class Level Mechanics

3.1 Research Problem

This two-partite case study examines the role of dependency propagation in the accumulation of technical debt for software implementations. We approach the research problem by dividing it into two objectives. Objective I is to identify and produce a structured documentation for technical debt in the target implementation. Objective II is to understand the role of dependency propagation in the formation of this structure.

Fulfilling objective I requires that we are first able to distinguish between modifications made to advance the implementation and modifications made to repair the implementation. After identifying modifications that belong to the latter —and count as paying of technical debt, further information is required to identify relations between these modifications. Revealing the relations allows us to arrange the individual modifications to form a structure that indicates how technical debt has accumulated in the implementation.

Objective II is to understand the role of dependency propagation in the formation of this structure. Dependencies are formed between elements of the implementation. These elements and the rules for dependency formation between them are defined by the programming paradigm as well as the programming language. As each identified modification operates on a set of implementation elements, we can utilize the dependency formation rules to identify all elements that are dependent onto this set. Comparing the revealed dependencies to the connections in the technical debt accumulation structure is used, in this case study, to examine the role of dependency propagation in the accumulation of technical debt for the software implementation.

3.2 Target System

The system on which we will conduct the first part of this case study is called ViLLE. It is a collaborative education platform that is being developed and researched at the University of Turku [17]. During the nine years of its development, ViLLE's implementation has over gone several rehauls. The latest of which aimed at reducing end-user requirements to a bare minimum by converting the system into a SaaS (Software as a

Service). The research and development team simultaneously wanted to serve a broader spectrum of education subjects through extending the set of available exercise types. The old legacy exercise system was found to be too rigid for this purpose, and it was decided that this part of the system was to be refactored.

The first author has taken part in this process, and it has also been the focus of a thesis [10]. The thesis has documented the entire project, and it is consulted here to establish what parts of the system were targeted, what are the tools and practices used for the refactorization, what are the motivations as well as the requirements for the refactorization, and finally access to the version control system to conduct a retrospective of the refactorization.

The conversion to SaaS was made with the Vaadin web-application framework, for which development is carried out with the Java language. At the time of the refactoring the running configuration of the ViLLE system was comprised out of 122k physical lines of code, organized into a hierarchy of 26 Java packages, encompassing a total of 460 Java classes. Further analysis in [10] pinpointed the exercise system problems to four Java classes. These core system classes were responsible for the execution, modification, storing and retrieving, as well as modeling of interactive exercises in ViLLE. For each of these, [10] captured a set of problems as well as a set of reparative actions which were used as the starting point for the refactorization.

The refactorization was carried out independent from ViLLE system's daily development. In practice, a separate version control branch was used which only contained commits that corresponded to meeting the requirements of the refactorization. From the perspective of providing data for the first part of this case study, we interpret the previous as follows: all modifications observable from the mentioned version control branch constitute paying off technical debt.

3.3 Data Collection and Analysis

The data collection was started by constraining to the version control branch identified in Section 3.2. As this constriction limited the data set to only containing modifications that corresponded to refactorizations, we could proceed to build the structured representation for technical debt accumulation for this implementation (see Section 3.1).

Section 2.1 discussed how technical debt manifests in software implementations: reliance onto technically incomplete objects may call for adaptation in the dependents. Successfully paying off technical debt for the implementation implies that individual refactorizations are able to nullify the adaptations as well as to remove the root cause. In this case the root cause was confined within four Java classes (Section 3.2). Each of these classes was responsible for implementing an independent and distinctive functionality in the system. As the structured representation for technical debt accumulation was to reflect how inabilities in implementing system functionalities had affected the system, four root nodes were chosen. Each root node consisted out of a set of modifications corresponding to all refactorizations made to repair the functionality in —and to remove the root cause from— one of the aforementioned root cause classes.

Having identified the root nodes and their modification sets, we continued to study the remaining modifications. Links between modifications were determined as cause-effect-relations: a link existed between modifications if successful completion of the cause-one required a successful completion of the effect-one. That is, they shared a common modification context. The chronological order —of cause-modifications taking place before effect-modifications— was ensured by observing that the effect-ones could only exist in revisions commits that were the same or superseded that of the cause-ones'. The two step process was repeated until all modifications were associated with the structure for technical debt accumulation.

To facilitate the fulfillment of the second objective, we related information about the propagation of dependencies to the structured representation for technical debt

accumulation. As the system in question is implemented using the Java language, the object-oriented paradigm as well as the Java technology can be consulted for information about the propagation of dependencies in the implementation. Applying this, for each modification, the set of implementation elements dependent onto the modification's target element were identified. This set was then queried to find out if it contained elements that are targets of other modifications which are linked to the original modification; used to spawn the set. The propagation identification results were then associated with the structure for technical debt accumulation to indicate the role of dependency propagation in its formation. Analysis of the final structures in the following section fulfills the second objective.

4. Results

An interval of ViLLE's implementation revisions, discussed in Section 3.2, was analyzed for the first part of this case study. We found that the refactorization consisted out of 140 individual modifications, or refactorizations, which affected a total of 71 Java classes. Amongst these were the four Java classes encompassing what [10] had identified as the root cause. Observing which modifications realized the removal of the root cause in these four classes lead to the formation of four modification sets that served as the root nodes for our structured representation for technical debt accumulation. An iterative process of identifying cause-effect-relations, according to the case study design (see Section 3.3), lead to populating the four substructures with the remaining modifications. Identification of cause-effect-relations for all modifications also indicated that a modification could only be associated with a single substructure.

The resulting technical debt accumulation structure was then associated with information regarding the propagation of dependencies. This corresponded to identifying the target elements for all modifications, identifying sets of elements that were dependent on the target elements, searching for possible relations between element dependencies and modification links, and finally relating this information to the technical debt accumulation structure. The resulting structure is depicted in Figure 2 and presented in the following as four Technical Debt Propagation Trees (TDPT).

4.1 Technical Debt Propagation Trees

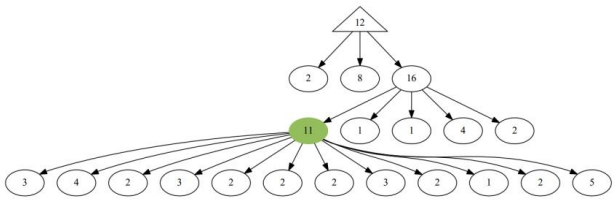
Figures 2a through 2d depict the resulting Technical Debt Propagation Trees when modifications made to Java classes responsible for execution, modification, storing and retrieving, as well as data modeling the exercises in the ViLLE system are respectively used as root nodes for the analysis presented in Section 3.

The same visual aids apply for all presented TDPTs. Nodes represent modifications. Arrows indicate cause-effect-relations between modifications. The root node —the used modification set— is modeled as a triangle. If a dependency exists between the target elements of modifications of a cause-effect-relationship then the node for the effect-modification is modeled as an ellipse. If not, the node is modeled as a rectangle. If the modification type is addition of new implementation elements then the node is colored green (light shade). Else, if the modification type is removal of implementation elements then the node is colored red (dark shade). Finally, the number inside each node is the sum of dependencies to modifications' target elements.

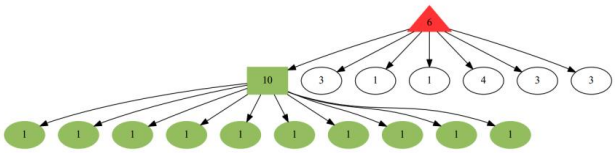
4.2 Analysis of the Technical Debt Propagation Trees

In analyzing the TDPTs (see Figures 2a through 2d), we have observed the following. First, modifications to implementation elements with a large number of incoming dependencies seem to invoke an increased number of further modifications. However, this is not consistent as the number of incoming dependencies deviates from the number of invoked modifications (see Figure 2d).

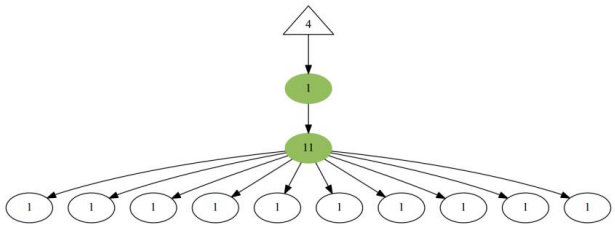
Second, examining the cause-effect-relations forming the edges of the TDPTs, in all but two cases there exists a dependency between underlying implementation elements for an observed cause-effect-relationship between modifications. Close examination of the first non-dependency case (between the root and a second tier node in TDPT in Figure 2b) revealed that here refactoring separated functionality from the original area, and the newly formed element hierarchy was thus made completely independent from its original element, leading to non-dependency between modifications' target elements. In the second non-dependency case (between the root and a second tier node in TDPT for data modeling in Figure 2d) similar motivation could be observed: exercise type declarations were separated here from the generic exercise data model and placed into their own containing class. Hence, in almost all cases dependency propagation was the apparent cause for technical debt accumulation.



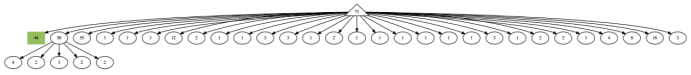
(a). TDPT Having Modifications Made To Ville's Exercise Execution Implementation as Its Root Node



(b). Tdpt Having Modifications Made to Ville's Exercise Storing and Retrieval Implementation as Its Root Node



(c). Tdpt Having Modifications Made to Ville's Exercise Modification Implementation as Its Root Node



(d). Tdpt Having Modifications Made to Ville's Exercise Data Modeling Implementation as Its Root Node

Figure 2. ViLLE Refactorization TDPTs

Third, examining the depths of the TDPTs: in the case of TDPTs for storage and retrieval (Figure 2b) as well as data modeling (Figure 2d) the tree depth is three, while for TDPTs for execution (Figure 2a) and modification (Figure 2c) the tree depth is four. Further, for all leaf modifications the number of dependencies incoming to their target elements is rather low, under ten. Except for the few cases mentioned previously.

Fourth, observing the tree structures: in the case studied system, modifying a component that is responsible for providing a data model in the implementation (see TDPT for data modeling in Figure 2d) seemed to invoke a series of modifications that could be described as shallow but wide. While, modifications responsible for implementing specific features of the system seemed to invoke a series of modifications that were more narrow and focused than the former (see TDPTs for execution, modifying, and storing and retrieval in Figures 2a, 2c, and 2b respectively). This seems to indicate that for the refactored-to-be elements of the implementation, their role in the system could be used to postulate the course of the refactorization undertaking in this part of the system.

5. Conclusions and Validity

Analysis of the ViLLE system's TDPTs in Section 4.2 observed the following, with a few exceptions. First observation (*O1*), number of incoming dependencies correlates with the number of propagation paths (as per Bianchi, *et al.*, [3] in Section 2.2) for technical debt. Secondly (*O2*), dependency propagation can be seen to drive the accumulation of technical debt in this software implementation. Thirdly (*O3*), examination of the TDPT depths supports what McGregor, *et al.*, [14] hypothesized about technical debt diminishing due to dependency propagation. Finally (*O4*), the role of a system component could be used to explain how technical debt had propagated in the system.

Concluding onto observations *O1* - *O4*: it is evident that dependency propagation plays a significant role in the accumulation of technical debt for the software implementation in question. The propagation of dependencies, which is possible to explicitly indicate for a software implementation, can be used to predict the size and distribution of technical debt. Also, successful derivation of conclusions indicates that the approach derived for this case study is viable for examining the role of dependency propagation in the accumulation of technical debt.

As this case study examines a unique phenomenon in a specific context, applicability of presented results requires that certain threats to validity are addressed. A matter affecting the validity of the case study's construct is the definition used for an acceptable modification. For the ViLLE system, all observed modifications were accepted as paying off technical debt since [10] described them as instances of refactorizations (following the definition given in Section 2.1), and the data set was explicitly and accordingly limited off (see Section 3). Still, it can be argued that the used acceptance criterion was too loose, and the resulting TDPTs were over populated. A counterclaim to this is that the case study specifically targeted a refactorization project, with the foremost intends of not altering the system's behavior, and as such the possible bias should be very small in size.

Deriving results in the first part required that we identified a causal relation between the propagation of dependencies and the accumulation of technical debt. Matters distorting this affect the case study's internal validity. Section 3 explained the processes used for determining both cause-effect-relations between modifications as well as the propagation of dependencies between implementation elements. Here, the latter is determined based on static rules and confirmed in the program's ability to function. However, determining of cause-effect-relations was based on the researchers' ability to distinguish if two modifications shared a common context. While most information in the contexts —for example, close chronological ordering and shared implementation areas— lead to a strong conclusion, the possibility of making a wrong decision cannot be excluded. However, issue-free and successful association of all modifications indicates that uncertainty played a small role in this step.

6. Second Case - Class Member Level Mechanics

Taking the conclusions made in Section 5, we wanted to verify if similar mechanics could be observed for a separate data set and for a lower abstraction level in the implementation. Eclipse is an open source integrated development environment (IDE) implemented in the Java language. Easily accessible source, a well-documented implementation history, and application of common development techniques makes this a good candidate for the second part of this study.

In order to produce comparable results, we overcome the same objectives described in Section 3.1. We first identify and produce a structured documentation for technical debt in the target implementation for which we then determine the role of dependency propagation in its formation. Overcoming the first objective is relatively easy in comparison to the first part of the case study. Due to Eclipse's well documented development history, we can derive explicit structural representations for its technical debt by first querying its bug database for prominent candidates. For these chosen non-conformalities, we then commit a retrospective analysis in the Eclipse version control system. In here, all bug fixes are recorded as individual entries tagged with the identifier of the corresponding bug. Thus, a technical debt instance's root can be determined from the bug report while a chronologically ordered set of tagged version control entries is queried for the spawned modifications.

The second objective is overcome in a fashion identical to what was described in Section 3.3: for each technical debt instance, we take both the root cause and its reparative modifications and consult the Java language specification in determining if dependencies exist between these sets. Combining the results from the first and the second objective, we form TDPTs for the chosen bugs.

6.1 Execution

The Eclipse system comprises a vast software implementation. In our other study [12] for a cohesion visualization approach, we were able to distinguish over 91 thousand class members with over 1.5 million unique dependencies, from the Eclipse's core implementation alone. As the bug database manifests similar magnitudes, we need a way to limit off and choose representative instances from it. Figure 3 represents the query used to find non-conformalities —or bugs. Noteworthy here is that we limited the query to Eclipse's version 3.0, because it is a stabilized release, and the bug state to *verified fixed*, to ensure that correct reparative action had been found and taken to overcome the issue.

Classification:	Eclipse
Status:	VERIFIED
Resolution:	FIXED
Version:	3.0
Component:	Compare, Core, CVS, Debug, IDE, Resources, Runtime, Search, SWT, Team, Text, UI
Product:	JDT, Platform
Keywords:	Accessibility, consistency, performance, security, usability
Changed:	(is greater than or equal to) 2004-06-25
Creation date:	(changed after) 2004-06-25

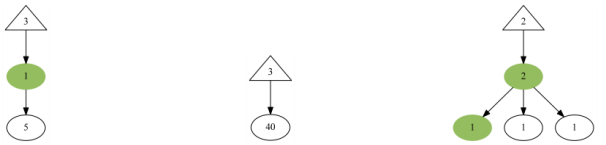
Figure 3. Used Query to the Eclipse Bug Database

We took a random sample (N=10) from the query in Figure 3. The sample representativeness was enhanced, by normalizing the results with distinct component bug counts. For each bug in the sample, a manual retrospective was conducted as per Section 6: for each bug, the version control system was queried for revisions carrying the corresponding tag or identifier. For all found revisions, modified class-members were identified based on comparing the revision at hand with its immediate predecessor. Validity of found modifications bases on Eclipse's commit policies which forbid any other commit content than what is tagged. And all bug fix commits are tagged only with the corresponding bug identifier. Lastly, the root node, to which all found modifications connect to, was identified for each bug from their initial reports: for Bugzilla bug reports the author is given a text field in which the initial problem location can be stated. This placed an additional restriction for our bug query: we can only consider bugs for which the initial problem location can be determined from the report. In cases where this was not possible, a new sample was taken (from the query in Figure 3).

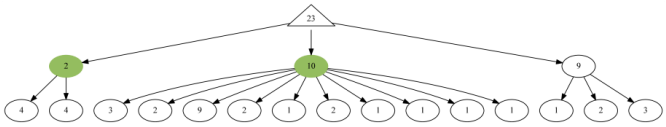
7. Resulting Technical Debt Propagation Trees

For each bug, combining found root locations with corresponding modifications, we form a Technical Debt Propagation Tree (TDPT). Figures 4a through 4j depict the TDPTs for the sample set queried in Section 6.1. These trees follow the visual notations defined in the first part of the study in Section 4.1.

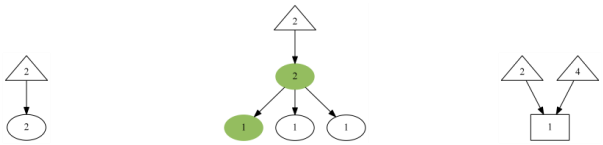
(a) Bug #73109 **(b) Bug #73110** **(c) Bug #73259**



(d) Bug #73950 **(e) Bug #73951** **(f) Bug #77491**



(g) Bug #78860



(h) Bug #82151 **(i) Bug #88065** **(j) Bug #148965**

Figure 4. Eclipse Bug TDPTs

Analysing the TDPTs, in Figure 4 we see that the number of inbound dependencies correlates with the number of captured technical debt propagation paths. In cases 4c and 4g, the correlation deviates from linear while the total accumulated modification count is substantially higher than for the rest of the cases. Second, in all but case 4j, there exists a direct dependency between target elements of subsequent modifications which indicates a strong involvement of dependency propagation in the accumulation of technical debt. Closer inspection for case 4j revealed that the originally speculated location for the bug was only a manifestation channel which resulted in reparation in, a close but, unrelated area.

Thirdly, for all TDPTs in Figure 4 we observe the lowest and the highest tree depth as one and three respectively. Since the leaf nodes still have inbound dependencies, this indicates non-forced diminishment for technical debt. Fourth, notions regarding tree shape need to be made whilst wary of the bias caused by their small size. Nevertheless, almost all trees demonstrate a shape that is more deep and narrow, consistent with bugs generally being reparations for very specific and focused functionalities. In cases 4c and 4g, the wider branches result from modifications to low level data models: variables retaining data for a class state, to which majority of its members depend on.

8. Conclusions and Relation with the Former Part

In the first part of this case study the accumulation of technical debt was retrospectively analysed for a refactorization project and it resulted to four observations *O1* - *O4* (see Section 5). The second part pursued verifying similar notions for an independent data set, at a lower abstraction level. The results for this were analysed in Section 7, and they allow the following to be concluded —while respecting the inconsistencies discussed in Sections 4 and 7.

Observation *O1*, the number of incoming dependencies correlating with the number of technical debt propagation paths, applies for the second data set as well. This leads us to presume that technical debt can be observed and tracked at various abstraction levels, as long as the level is capable of providing a mechanism for observing inter-component relations within it. Also, high correlation increases trust in each link's capability to carry technical debt, and, thus, allows easier application of link structure analysis approaches like the PageRank algorithm [16], to rank and assess technical debt volatilities for nodes

in captured call graphs. Observation *O2*, of dependency propagation driving technical debt accumulation, is also applicable for the second data set. This relation was actually stronger for the second part, and seems to indicate that at the lowest abstraction levels dependency propagation is solely responsible for the accumulation of technical debt.

A strong observation of *O3*, of technical debt diminishing due to propagation, was also made. This provides further verification to earlier assumptions regarding the natural constriction applied by host class boundaries. Finally, observation *O4*, of component role explaining the shape of the TDPT, was partially made for the second part. The small tree size is obfuscating, but this observation held especially for data structure modifications.

In conclusion, these observations indicate that software implementation technical debt manifests abstraction level independent characteristics that adhere to the mechanics of dependency propagation but with certain exceptions. Further analysis of similar cases at differing abstraction levels should capture these exceptions in a static model that can be used in providing automatic updates for captured technical debt information.

9. Future Work

Research following this case study will build on the conclusions presented in Sections 5 and 8. Firstly, as per Flyvbjerg [6] and Yin [21], we note that two case studies are not adequate for theorem forming, but they serve as grounds for generalization. Hence, we intend to employ the approach derived and used in this case study for additional data sets. Improvements, like automating tree depth calculation, will be introduced to help overcome larger data sets. We expect this to provide more details on the intrinsics of technical debt accumulation in software implementations, in addition to further examining the role of dependency propagation in this process. We are especially interested in identifying if certain dependency types accumulate technical debt differently, if the role of system components can be used to further explain the size and distribution of technical debt, and if other mechanisms can be established for the non-dependency driven accumulation of technical debt.

Further, the results of this, and following analyses, will be used to build and assess the propagation model used by the DebtFlag-tool [9]. As the tool relies on the ability to maintain technical debt notions through this model, explicitly presenting the differences in the propagation paths of technical debt and dependencies between implementation elements will allow for further enhancements. As such, our ongoing research is focused on assessing and evaluating possible models to identify viable solutions. A strong candidate is the PageRank link structure algorithm by Page et al. [16]. Initial analyses, with the data provided in this article, have yielded promising results especially in accommodating the diminishment characteristic of technical debt.

Acknowledgements

The work of Johannes Holvitie is supported by the Nokia Foundation, the Ulla Tuominen Foundation, and the Finnish Foundation for Technology Promotion.

References

[1] G. Avellis, “Case support for software evolution: A dependency approach to control the change process”, In Computer-Aided Software Engineering, Proceedings, Fifth International Workshop on IEEE, **(1992)**, pp. 62–73.

[2] L. A. Barowski, J. H. Cross, *et al.*, “Extraction and use of class dependency information for java”, In Reverse Engineering, Proceedings, Ninth Working Conference on IEEE, **(2002)**, pp. 309–315.

[3] A. Bianchi, D. Caivano, F. Lanubile and G. Visaggio, “Evaluating software degradation through entropy”, In Software Metrics Symposium, METRICS, Proceedings, Seventh International, IEEE, **(2001)**, pp. 210–219.

[4] 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.
[5] W, Cunningham, "The WyCash portfolio management system", In Addendum to the proceedings on Object-oriented programming systems, languages, and applications (OOPSLA), vol. 18, **(1992)**, pp. 29–30.
[6] B. Flyvbjerg, "Five misunderstandings about case-study research", Qualitative inquiry, vol. 12, no. 2, **(2006)**, pp. 219–245.
[7] M. Fowler and K. Beck, "Refactoring: improving the design of existing code", Addison-Wesley Professional, **(1999)**.
[8] Y, Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. B. Da Silva, A. L. M. Santos and C. Siebra, "Tracking technical debt - an exploratory case study", In Software Maintenance (ICSM), 27th IEEE International Conference on IEEE, **(2011),** pp. 528–531.
[9] J. Holvitie and V. Leppänen, "DebtFlag: Technical Debt Management with a Development Environment Integrated Tool", In Managing Technical Debt (MTD), 2013 Fourth International Workshop on IEEE, **(2013)**.
[10] J. Holvitie, "Code level agility and future development of software products", Master's thesis, Department of Information Technology, University of Turku, **(2012)**.
[11] J. Holvitie, M.-J. Laakso, T. Rajala E. Kaila, and V. Leppänen, "The role of dependency propagation in the accumulation of technical debt for software implementations", In 13th Symposium on Programming Languages and Software Tools, **(2013)**, pp. 61-75, University of Szeged.
[12] J. Holvitie and V. Leppänen, "Illustrating software modifiability - capturing cohesion and coupling in a force-optimized graph", In 14th IEEE International Conference on Computer and Information Technology (CIT 2014), **(2014)**.
[13] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman and F. Shull, "Organizing the technical debt landscape", In Managing Technical Debt (MTD), Third International Workshop on IEEE, **(2012)**, pp. 23–26.
[14] J. D. McGregor, J. Y. Monteith and J. Zhang, "Technical debt aggregation in ecosystems", In Managing Technical Debt (MTD), 2012 Third International Workshop on IEEE, **(2012)**, pp. 27–30.
[15] I. Ozkaya, P. Kruchten, R. L. Nord and N. Brown, "Managing technical debt in software development: report on the 2nd international workshop on managing technical debt", held at ICSE 2011, SIGSOFT Softw, Eng, Notes, vol. 36, no. 5, **(2011)**, pp. 33–35.
[16] L. Page, S. Brin, R. Motwani and T. Winograd, "The pagerank citation ranking: Bringing order to the web", Technical Report, vol. 66, **(1999)**.
[17] T. Rajala, M.-J. Laakso, E. Kaila and T. Salakoski, "ViLLE: a language-independent program visualization tool", In Proceedings of the Seventh Baltic Sea Conference on Computing Education Research, vol. 88, **(2007)**, pp. 151–159, Australian Computer Society, Inc.
[18] M. P. Robillard, "Topology analysis of software dependencies", ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 17, no. 4, **(2008)**.
[19] C. Seaman and Y. Guo, "Measuring and monitoring technical debt", Advances in Computers, vol. 82, **(2011)**, pp. 25–46.
[20] 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 Managing Technical Debt (MTD), 2012, Third International Workshop on IEEE, **(2012)**, pp. 45–48.
[21] R. K. Yin, "The case study crisis: some answers", Administrative science quarterly, **(1981)**, pp. 58–65.

Authors

Johannes Holvitie, M.Sc. (Tech.), is a doctoral candidate of the Turku Centre for Computer Science and the Faculty of Natural Sciences at University of Turku. Mr. Holvitie's research focus and doctoral dissertation topic is *Technical Debt as a Factor in Efficient and Sustainable Software Development*.

Ville Leppänen, PhD, works currently as a software engineering professor in the University of Turku, Finland. He has over 100 scientific publications. His research interests are related broadly to software engineering ranging from software engineering methodologies, practices and tools to security and quality issues and to programming language, parallelism and algorithmic design topics.

Paper IV

IV

Illustrating Software Modifiability–Capturing Cohesion and Coupling in a Force-Optimized Graph

Holvitie, Johannes and Leppänen, Ville (2014). In *Computer and Information Technology (CIT), 2014 IEEE International Conference on*, pages 226–233. IEEE

2014 IEEE International Conference on Computer and Information Technology

Illustrating Software Modifiability - Capturing Cohesion and Coupling in a Force-Optimized Graph

Johannes Holvitie, Ville Leppänen
TUCS - Turku Centre for Computer Science &
Department of Information Technology, University of Turku
Turku, Finland
{jjholv,ville.leppanen}@utu.fi

***Abstract*—Software visualization aims to provide a more human-readable interface for the various software system aspects and characteristics. As majority of the time spent on modifying software is spent on gaining an understanding of an intangible and virtual system, the area of software visualization is widely researched as a solution to this. The paper in question presents a program visualization approach that focuses on illustrating the two software modifiability characteristics of cohesion and coupling. Unlike other approaches, which provide a visual representation for precalculated values, it uses the underlying cohesion and coupling mechanics to derive the actual layout. This allows the user to perceive the entire structure that has resulted to the cohesion and coupling values present in viewed nodes. There are three distinct steps to our approach. 1) Semantic analysis is used to record the static program structure into a directed and weighted graph. 2) The graph is then laid out using force-optimization to highlight important implementation structures. Finally, 3) sub-graph separation and further visual aids are provided to aid the user in observing cohesion and coupling for specific areas. Discussed benefits for this approach include information production efficiency, the ability to quickly analyze even large software implementations and intuitiveness of the visual delivery method.**

I. Introduction

Increasingly complex hardware has implicitly allowed more complicated software to be ran on it. This, in addition to the popularity of modern document-light, iterative and incremental software development methods, has created a challenging ground for software development tasks that depend on information regarding the software implementation's state. The likes of iteration planning and software maintenance are forced to deal with high levels of inaccuracy as they manage large, self-emergent, and intangible software systems.

Software visualization aims to increase tangibility by way of utilizing graphical illustrations to depict varying structures within the software [1]. The aim is to facilitate both human understanding as well as effective use of the illustrated parts [2]. Program visualization is a sub-method of the former. It works with static and dynamic data to provide views into software implementations' structural and functional properties respectively. Program visualization is fundamentally an information retrieval method.

Software process components that allocate resources either for reparative or function additive actions are interested in how the current implementation is capable of accommodating modifications. Cohesion and coupling are statically assessable measures of modifiability [3]. A number of metrics, e.g. [4]–[8], capture them but they predominantly produce results which require combination and prolonged analysis in order to reach their full potential. Hectic software development environments can find this discouraging.

This paper introduces a novel program visualization approach that captures information regarding software implementation's state through utilizing the cohesion and coupling mechanisms. The main goal of the presented approach is to serve software projects with an efficient, intuitive and easily attainable medium that allows communicating about the state of software and especially its capability to accommodate modifications.

There are three steps to our approach: semantic code analysis, graph layout, and sub-graph separation. In the semantic analysis we traverse the source code of a program and form a set of Abstract Syntax Trees (AST). These ASTs capture program element interaction at the lowest abstraction level. Through filtering the ASTs we produce data on which of the elements call one another and how many times. As cohesion and coupling are indicators of program element interdependency measured through their relations, we transform the filtered call information to a sparse matrix—a basic form for a graph.

The second step, the graph layout, takes the sparse matrix graph and applies a force-directed layout algorithm to it. In the graph, program elements are represented by nodes and directed weighted edges connecting them represent the elements' call directions and frequencies respectively. In force-directed layout, the edges are perceived to be forces which influence the nodes. The layout is complete when a minimum energy state has been found for the graph. As the forces in this graph represent cohesion and coupling meta-information, the produced layout consists from structures that capture and highlight modifiability characteristics.

In the sub-graph separation part the user may query the large system-wide graph with program elements to produce sub-graphs that highlight the cohesion and coupling structures to which these elements belong. The sub-graphs are built by identifying the shortest non-weighted path for each element pair in the query set. The final sub-graph consists from all encountered shortest paths and the nodes immediately adjacent to these. We call these adjacent nodes context nodes as they deliver information on the query's neighborhood. The user can toggle displaying them.

Rest of the paper is structured as follows. Section II

978-1-4799-6239-6/14 $31.00 © 2014 IEEE
DOI 10.1109/CIT.2014.112

reviews background literature introducing cohesion and coupling, program visualization and applicable graph processing algorithms as well as our previous work. Section III introduces the approach by defining used graph components, formation of the graph and its layout, and querying for sub-graphs. Section IV demonstrates the approach by applying it for a large open-source software product. Received graphs are analyzed, discussed and evaluated in Sections V, VI and VII respectively. Finally, Section VIII concludes this paper and makes some sights into future.

II. Related Work and Background

In this paper we introduce a program visualization approach that applies force-directed layouting to a graph containing cohesion and coupling details for a software system. The first subsection here defines cohesion and coupling to provide a reference point for the approach's graph formation (see Section III-A). The second subsection derives requirements for the visualization in discussing related work on program visualization. The third subsection describes the chosen graph processing algorithms and other considered options. Finally, the last subsection presents our previous work to which the approach herein is an extension.

A. Cohesion and Coupling

The concepts of software cohesion and coupling were introduced by Stevens et al. in [3]. Coupling is defined as the measure of module interdependence. A high degree of coupling for a module indicates that its functionality is heavily dependent on the existence of other modules. A modification in these can be expected to cause a modification in the dependent. Reducing coupling minimizes the propagation of modifications. If modules are built from elements, then we need to minimize the relations between elements not in the same module. Cohesion captures this in measuring the interdependence of elements within the same module. High cohesion for a module indicates that elements forming it are together well capable of implementing its functional requirements—without outside assistance.

The rather abstract definition of cohesion and coupling has lead to the emergence of several capturing metrics for them. Many of them are not exclusive as they capture cohesion and coupling with slightly different characteristics. For capturing cohesion, the most well known metrics are the six versions of LCOM (Lack of Cohesion in Methods). Versions zero through three, presented in [4], [5], [6] and [7] respectively, capture lack of cohesion as the volume of those member functions that do not share a common variable. In LCOM4 member classification is ignored and cohesion is measured as the number of inter-function calls and the variables they share [7]. Finally, LCOM5 measures cohesion as the number of functions assessing variables [5].

Two notable coupling metrics are components to the Instability metric defined by Martin in [8]. Instability measures change resistance for a software component by calculating its Efferent (E_c) and Afferent Coupling (A_c)—the component's independence and responsibility respectively. Afferent coupling captures the number of outside components that depend upon modules within the component. Efferent coupling does the opposite in measuring the number of foreign modules required by the target component.

B. Program Visualization

Caserta and Zendra argue in their survey of static program visualization approaches [9] that *graphs* have the most suitable visualization characteristics for architecture-level illustrations of source code snapshots. They add that for large systems graph occlusion and edge congestion are possible challenges. From the surveyd approaches the *CodeCity* [10] is the only one utilizing cohesion or coupling for layout placement. While this is seen to be intuitive and effective, drastic changes caused by snapshot updates are source of confusion.

Langelier et al. [11] discuss a visualization approach for software development and maintenance. While also acknowledging the role of cohesion and coupling in the process, they conclude by claiming that for target analysis a hybrid approach of efficient automated visualization and human interaction for context limiting is a good compromise. In [12] Koschke lists problems relating to maintenance and re-engineering related software visualization. Most notable ones from this list are large graph size, possibility of filtering, software evolution driven visualization updates and taking node and edge semantics into account for automatic layouts.

C. Graph Processing

Finding global minima for a force-directed graph is an extremely difficult process with no proven solutions [13]. There however exist methods with the ability to produce "good" results. These approaches generally use a multi-algorithm or -level approach. An example of a multi-algorithm approach is presented in [14]. While, the multi-algorithm ones produce exemplar results, they suffer from high time and space complexities. Yifan Hu's multilevel algorithm [13] is an example of the latter. Using graph coarsening, initial layouting and then refinement, the algorithm produces quality results with acceptable complexity.

For our layout, we use the continuous force-directed layout algorithm ForceAtlas2 [15]. It has adapted its energy model from the energy models proposed by Fruchterman and Reingold [16] as well as Noack [17]. The complexity is decreased from $O(n^2)$ to $O(n * ln(n))$ by applying Barnes Hut [18] approach for force approximation. According to tests conducted by the algorithm's author, ForceAtlas2 fares well against the notable multilevel layout algorithm by Yifan Hu [13]. Yifan Hu's multi-level algorithm would have been used in this work, if not for its inability to consider edge weights.

As stated, after converting the implementation structure into a graph, we may utilize link structure analysis in order to calculate global importance ranking for its nodes. We don't consider query specific ranking algorithms here as similar information is already produced by our approach through the query based sub-graphs. One of the most used and studied non-query ranking algorithm is the PageRank by Page and Brin [19] and there exist several adaptations of it for the software context. Many of these have found applying PageRank to being especially useful in change impact analysis for indicating globally and locally important elements. Calculating PageRank values for nodes in our graphs allows us to carry similar

information to the queried sub-graphs—in addition to the visual information.

D. Previous Work

In our previous work we have introduced a mechanism for capturing notions about technical debt and displaying them at both the software implementation level as well as the project management level [20]. The tool that we have implemented for this mechanism is an Eclipse development environment plug-in called DebtFlag. The plug-in allows the use of different propagation models in order to support technical debt accumulation for implementations where the accumulation process differs due to e.g. the used implementation technique. We have also conducted a case study on a refactorization project in order to explore the various capabilities of technical debt accumulation and the role of dependency propagation in realizing them [21]. In this, we noted that the number of incoming dependencies for a module correlates with the number of modifications it invokes thus indicating that dependency propagation is a driver in technical debt accumulation, technical debt diminished due to dependency propagation and that the role of a system component explains, to a certain extent, the size and distribution of technical debt. In addition to this, we have also studied cohesion metrics in [22].

III. Approach

In this section we give an abstract description for our program visualization approach. We start by defining the graph components that present program elements and their relations. This is followed by a description for the semantic analysis procedure used to capture relations from the target program and to populate the graph. The complete graph is then laid out using force-optimization to visualize the information carried in its directed and weighted edges. Finally, queries can be made to the complete graph in order to separate sub-graphs to examine areas of interest.

A. Graph Components

Section II-A introduced cohesion and coupling as indicators of program complexity that base on information regarding the interdependence of individual program elements. In the first step of our approach this information is captured in a graph. The graph nodes represent the program elements and edges represent their interactions. Regarding the graph's input into force-optimization, actual information is carried as node and edge weights. As per the definitions of cohesion and coupling, this information constitutes only the degree of dependence between each program element pair. This leads to the following definitions for the graph components.

A node is a representation of a program element at an abstraction level in which explicit dependencies are formed. The abstraction level is dependent on the source implementation's paradigm and technique. For example, the class member level is considered for the object-oriented paradigm. In object-oriented implementations dependencies are formed against interfaces, interfaces are defined as classes, and classes consist from methods and variables. Hence, the program elements captured for an object-oriented implementation are the interface-forming methods and variables (applied in Section IV). Further, we note that the definitions of cohesion and coupling do not differentiate between types of software elements. From the perspective of force-optimized layout this means that all the nodes capturing the program elements should exert a uniform force. Hence, the nodes are left with equal weights and they only carry the program element's name as a unique identifier.

Edges capture relationships for all nodes in the graph. Cohesion and coupling are proportional to the number of dependencies between program elements. Interpreting this as a force-optimization problem means that for all program element pairs, the edge weights between nodes representing them is equal to the number of references between the elements. As both elements in a pair may invoke the other, the edges are directional to capture the two-way connection. When input to force-optimization, the total force between a node pair is the sum of directional weights for edges that directly connect them.

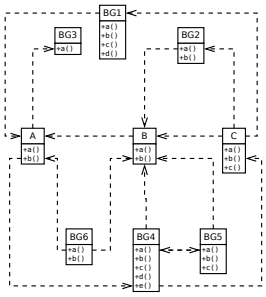
As a special case, a program element may refer to itself. Here the modeled edge has the same node in both ends. From the perspective of cohesion and coupling this type of reference has no value as it does not affect how the element connects to the surrounding system. Our approach takes this into account by default. Since the node weights are uniform and the edge capturing the self-reference is a loop, there is no observable force outside the node. Hence it can not be taken into account by force-optimization. This does however become an issue when applying link structure analysis. For example, the PageRank algorithm distributes node rank according to outbound edges. This values the rank of a self-referencing node a bit higher. As a remedy, loops with length one can be ignored when calculating these rankings.

B. Software Implementation Graph and Layout

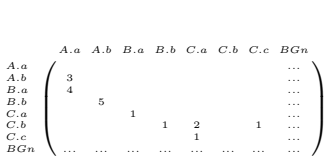
Forming the software implementation graph considers limiting off the implementation area, setting up a semantic program code analyzer and laying out the complete graph. These matters are overgone in the following. To facilitate efficient application we have provided a solution to automate these steps after initial user input (see Section IV).

A software implementation graph is a static call graph where the directed and weighted edges record call directions and frequencies for the uniform nodes that represent program elements (as defined in Section III-A). Before forming the graph, we must dictate which parts of the system will be considered. Usually this is a trivial matter of drawing the line between modifiable and unmodifiable components. The division has a drastic effect on the graph composition as unmodifiable components usually consist of static libraries towards which most relationships are introduced. Leaving the unmanageable assets out focuses the visualization but provides a less realistic overall picture. Abstract description of the delimitation allows automatic classification of encountered components.

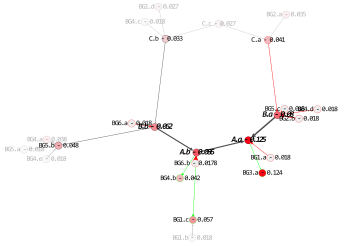
After the limitation we are left with a number of source code files encompassing the target implementation. Large size calls for automated approaches to determine relationships. A number of semantic analyzers exist for different programming languages to overcome this. The expected analysis output is an Abstract Syntax Tree (AST). The AST captures program element names and their types. This information can be used



(a) UML class diagram



(b) Relations captured in a sparse matrix



(c) Complete graph laid out with ForceAtlas2 and displaying a context query `B.a, B.b`

Fig. 1: Example program

to determine relations between them. Consulting the language's semantic specification [23] allows to distinguish the abstraction level discussed in Section III-A and to derive a library of possible element types. This library can then be used to transform the AST to a sparse matrix where column and row headers represent valid program elements and cells capture the number of directed relations between them.

The sparse matrix (see Figure 1b) is a complete graph lacking visual representation. To layout the graph, we use the ForceAtlas2 layout algorithm (see Section II-C). No user input is required for this step if the chosen default values for running the layout algorithm are accepted. In our approach we utilize the Gephi visualization engine and toolkit [24]. It allows us to convert the sparse matrix in to an in-memory graph and to automate its layout.

Finally, we may calculate a global importance ranking for the nodes in the graph in order to highlight interesting areas prior to context specific querying. PageRank is well supported by several static analysis tools as well as graph visualization tools. Figure 1c presents the final layout for the program in Figure 1a with corresponding PageRank values indicated for its nodes.

C. Context Querying for Sub-Graph Extraction

The graph formed in Section III-B can be used to make general observations but observing cohesion and coupling for a specific program element subset, a context, requires that it is separated from the system graph. Queries for the discussed element-level are supported as a default, since the graph nodes contain the element identifiers. Support for higher level queries requires that the approach is provided with semantic knowledge that can be utilized to convert the query back to the element level. After receiving the context query the separation is visualized by identifying relationships between the query's elements.

Relationships within a context are recorded as shortest paths between all possible component combinations that form the context. Use of the shortest paths approach is justified in that the graph formed in Section III-B is fundamentally a static call graph. The shortest found path between two elements is the least interfered demonstration of a relationship for them. In the case of several shortest paths existing for a pair they are all are considered. The weights and directions are considered irrelevant during this process firstly because the information is already present in the graph layout and secondly because reference frequency above zero is enough evidence to indicate existence of a relationship.

Figure 1c represents the complete graph with a query specific sub-graph extracted from it. The separated and highlighted graph is the result of applying the shortest path approach presented in the previous paragraph. The input to this approach encompasses a context containing two program elements `B.a` and `B.b`. In addition to the query itself, the highlight records all directly related elements for all paths. The related elements can be left undisplayed leading to a less obfuscated visualization but this leaves out the often interesting immediate neighborhood for this context. Carrying the neighboring nodes is always done at the cost of clarity.

For displaying the nodes' immediate neighborhood we use the following colors. For edges, shortest found paths are marked with black so as to clearly indicate relations within the queried context, those representing dependencies to context nodes are marked red and those representing dependencies originating from context nodes are marked green.

Now, the red edges capture the 'change group' for the query context. That is, if changes were made to nodes in the query context and the changes would not be contained within the nodes themselves, additional changes would propagate through the relationships indicate by the 'change group'.

Similarly, the green edges capture the possible 'root cause set'. This is the set to which the context nodes are directly dependent on to. For example, if a query is utilized to discover a cause for a problem within a certain context, then the 'root cause set' should also be considered as the functionality implemented in the query context is directly dependent on to and affected by this set. Further coloring, such as gradient coloring, for the nodes can be used to indicate superimposed rankings (like PageRank in Figure 1c).

Alpha blending (in Figure 1c) is used to fade out non-considered parts and to enable comparison between the sub-

graph and the host graph. Allowing relative distances to be observed for the sub-graph constitutes a major contribution for this paper. For a force-optimized graph that captures cohesion and coupling, observing small distances between its nodes conveys information about high cohesion for it. Similarly, at a higher level, if a sub-graph forms a hub that is clearly distinguishable from the host graph then the sub-graph has captured a context which should be loosely coupled.

IV. Application

We wanted to trial our approach on a large open source software project with a well documented development history. Due to previous experience with the Eclipse project it was selected. Eclipse Foundation develops an open-source integrated development environment (IDE) that supports editing and building a multitude of languages. In the following, we discuss composition of data, building a system wide and project specific implementation graphs, using bug reports as query contexts, and the extraction of context specific graphs.

A. Eclipse

The Eclipse platform architecture is built on the concept of plug-ins. Functionality is provided through them and they can be extended to introduce user-defined additions. The Eclipse foundation manages development of core plug-ins and divides them into sets called products. The core release of Eclipse with Java-language support encompasses two products: `JDT` and `Platform`. We consider the Eclipse release version 3.0 for our trials.

We capture cohesion and coupling in the Eclipse system by building a single large graph to encompass the entire system as well as smaller, more focused graphs. In building the graphs we follow the process described in Section III-B.

We utilize the services of our DebtFlag plug-in (Section II-D) to first discover all source components at the valid abstraction level. The plug-in uses Eclipse's AST processor to accomplish the task. Since Eclipse is implemented using the object-oriented Java language, we identify that all functionality is implemented in classes. Classes are described by their interfaces which are constructed from members of varying visibility. Java Language Specification [23] dictates that these can be either variables or methods. This corresponds to the sought after abstraction level and we capture it by generating a node in the graph for all such occurrences. The DebtFlag is utilized again to discover all direct use relations between any two discovered interface parts. All such occurrences are modeled as edges where the weight carries information regarding invocation frequency for this pair and this direction.

Applying this process for the entire Eclipse's version 3.0, yields us with a graph containing 121K nodes and 1.50M weighted directed edges between them. Iterative construction of this graph took 36 minutes when running 4 threads on an Intel Core i5-2410M @ 2.3GHz and 8GB RAM machine. After completion, the PageRank values can be calculated for each node to change their coloring to reflect this. Figure 2 presents this graph after 1421 iterations of the ForceAtlas2 algorithm (see Section III-B). The continuous algorithm was stopped after no movement was perceived between graph layout iterations. The layout took 17 minutes with the Gephi toolkit [24] when running 8 threads on an Intel Core i7-2600K @ 4.7GHz and 16GB RAM machine.

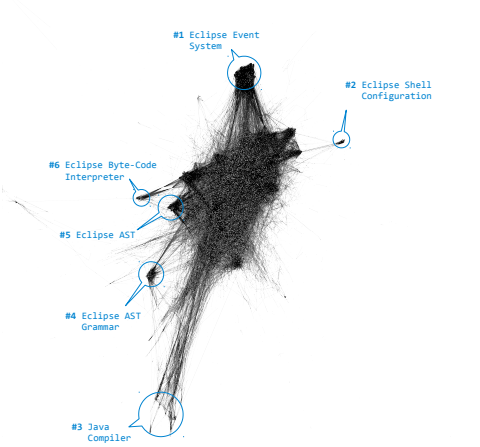


Fig. 2: Complete cohesion and coupling graph for the Eclipse system. Distinguishable hubs are highlighted

Since the system wide graph (Figure 2) is very large, we extract a smaller, more focused, graph encompassing a single project. For our example, we chose the *Debug* project, which is responsible for the `JDT.Debug` and `Platform.Debug` components. This graph is formed by extracting their elements from the system wide graph. Figure 3 contrasts the extraction against the system wide graph.

B. Extracting Sub-Graphs

Bugs represent an area for which resolving context cohesion and coupling is of especial interest as this communicates about the ease of chance for it. Constructing the graphs for Eclipse's version 3.0 allowed us to survey the Eclipse bug database in order to identify candidate bugs. In the following, we present an example bug, derive a context of interest from it and finally extract a sub-graph to present cohesion and coupling for this context.

In Section IV-A we formed a graph for the Eclipse Debug project. We query the Eclipse bug tracker for a bug declared for this project and for version 3.0. Bug number 148965 was chosen for this example. Initial documentation for this bug declares two problem elements `CompositeSourceContainer.findSourceElements(..)` and `PackageFragmentRootSourceContainer.findSourceElements(..)`. To identify cohesion and coupling for this context we proceed as described in Section III-C to form a sub-graph.

In this case, a direct connection exists between the two context elements resulting in finding a single shortest path with length one. Figure 4 displays the extraction with context nodes highlighted against the Debug project's graph in Figure 3.

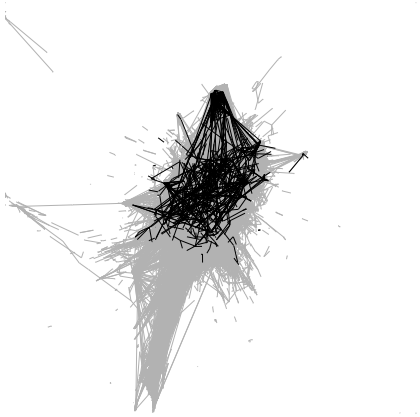


Fig. 3: Cohesion and coupling graph for the Debug project

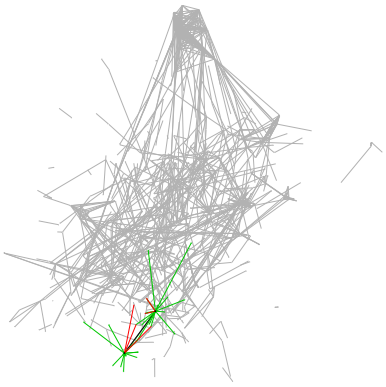


Fig. 4: Bug #148965 highlighted from the Debug project graph

V. Analysis

In Section III we introduced a process to form and layout a cohesion and coupling aware software implementation graph. In Section IV we applied this process for the Eclipse implementation to derive three graphs with increasing accuracy. In this section, we provide an analysis of the graphs in order to distinguish advantages and challenges related to the presented approach.

A. System Wide Graph

The system wide graph was formed in Section IV-A and depicted in Figure 2. Due to the vastness of Eclipse's implementation, we needed to combat against obfuscation (see Section II-B) when displaying it. This is done via applying a preview ratio: the layout is derived, as described previously, for all components but only a fifth of them are displayed. For dense graphs, this procedure retains global size and measure information while making the graph more approachable. When displaying query contexts in smaller graphs, this ratio may not be applied as position of every node carries valuable information.

Inspecting the received layout (in Figure 2), we first note that the graph consists from a number of hubs. Going over the nodes in the hubs, we note that single hubs capture program elements that are closely related: program elements are either from the same class or from a combination of classes responsible for implementing a shared functionality. This behavior is expected from a graph capturing cohesion and coupling.

There is high hub density in the main body of the system wide graph. Taking into account that the graph captures the core implementation for the Eclipse system we can expect a volume of nodes to be close to one another. The high density does however make the centre of the graph rather obfuscated and without dynamic highlighting or further node reduction it is very difficult to observe singular hubs in this area. We do however observe a number of hubs protruding from the centre mass.

Figure 2 has six of these outer hubs marked with numbers. Number one contains elements that form the Eclipse user interface and resource control event system. Second contains elements responsible for Eclipse's shell configuration. Third contains elements interacting with Java compilers. Fifth and fourth contain the Eclipse AST and its grammar respectively. Finally, sixth contains the Eclipse Java byte-code interpreter. All these hubs share a common trait in implementing a very specific functionality and we further argue that the hubs distance from the graphs centre correlates with the level of independence—that is high cohesion and low coupling—perceived for each hub.

B. Sub-Graphs

We extracted the Debug project from the system wide graph and presented the results in Figure 3. As mentioned, this project consists from two parts `JDT.Debug` and `Platform.Debug`. The former implements a language independent debugging model, where as the latter extends on this to provide Java debugging.

Form of the Debug project's graph can be explained as follows. The centre mass consist mainly from implementing the debugging tools in the user interface. Longer reaching edges capture queries to the AST, the process and memory controllers as well as configuration of the debugging shell. All these interface parts rely on Eclipse's interface event system to function and thus a large distinct hub exists at the top of the graph (see Section V-A). The project graph leads us to argue that the Debug project does not make unexpected references, it is tightly coupled with Eclipse's core implementation and this results in a low level of intra-project cohesion.

The context specific graph for Eclipse's bug no. 148965 is presented in Figure 4. The underlying graph is a scaled up version of the Debug project graph and the query can be seen

highlighted at the bottom. Surveying this graph, we note that the separated context is at the edge of the project graph. We would not automatically interpret this as the context being a less cohesive part of the project but it does lead to another observation.

Since the context is at the edge we can observe that some of the green edges could be coming from outside the project graph. Since the green edges indicate possible root cause elements (see Section III-C), this can mean that the actual reason for the bug in question is in another project and it is just first observed for an element in the Debug project. Fortunately, similar behavior can not be observed for the red edges. This means, that when the bug's elements are modified, further spawned direct changes in dependents are constrained within the project.

Lastly, a notion about the context's cohesion measures. Our example query consisted from two program elements `CompositeSourceContainer.findSourceElements(..)` and `PackageFragmentRootSourceContainer.findSourceElements(..)`. While their signatures as well as the found shortest path of length one indicate close relation, the geometric distance between the elements—relative to the project graph—remains large. Observing the geometric distance makes it evident that these two components, despite their similarities, are actually coming from two different plug-ins `Platform.Debug` and `JDT.Debug` and as such may require divergent context knowledge when modified.

VI. Discussion

The previous Section V made a number of observations based on visual analysis. Here we discuss their implications in more detail. Regarding the analysis of the system wide graph, we observed that the approach is capable of separating hubs that had distinct functional goals. This is an important initial indication of the approach's autonomous capability to highlight structures that are of interest from the perspective of cohesion and coupling. We demonstrate metric results for this in the next section.

Further, we were able to make observations regarding the Debug project's integration into rest of the system in addition to deriving additional information for a bug in the Debug project. These observations show that the single visual presentation used was capable of letting the user explore structures ranging from thousands to just singular program elements. However, use of the 'preview ratio' for the system wide graph indicates that, in case of a very large system, the presentation is prone to edge congestion which obfuscates the visualization. Improvements to this are currently pursued and some possibilities are discussed in future work.

Regarding the context queries, the resulting sub-graph for bug 148965 could be used to argue that 1) the possible root cause for this problem maybe coming from outside the hosting Debug project, 2) all modifications spawned from fixing the bug would be limited to the hosting project and 3) the geometrical distances between the context elements indicated that the bug encompassed elements that were not very closely related. All these observations were made based on visually available information. We interpret lower implementation technique and context knowledge requirements for the made observations as an indication of the approach's intuitiveness.

VII. Evaluation

We discussed our observations about the approach's ability to distinguish structures of interest from the perspective of cohesion and coupling. In Section V-A we presented the entire Eclipse system (in Figure 2) and we discussed our expectations of cohesion and coupling correlating with the six distinguished hubs. To provide initial evaluation for our approach, we calculate well established measures of cohesion and coupling for these hubs.

Table I records cohesion and coupling measures for each distinguished hub (see Figure 2) as LCOM4 and total couplings values. In addition to specific hub values there are the values for all resources in Eclipse. All versions of LCOM (introduced in Section II-A) produce inversed results: lower ones indicate higher cohesion. The value range of LCOM4 is $[1.0, \infty]$. The total couplings measure is the sum of afferent A_c and efferent couplings E_c.

TABLE I: Cohesion and coupling measures for hubs

Hub	#	LCOM4	Couplings
Eclipse Event System	1	1,0583333333	41
Eclipse Shell Integration	2	1,0	39
Java Compiler	3	1,0	259
Eclipse AST Grammar	4	1,0333333333	47
Eclipse AST	5	1,35	49
Eclipse Byte-Code Intrp.	6	1,0230769231	44
Mean for all resources		**1,0606837607**	**36,3646723647**
St.dev. for all resources		**0,171359571**	**49,7891359905**

Inspecting the LCOM4 values for hubs in Table I we first note that most of them are below the average and very close to the bottom value of 1.0. This would seem to indicate that these hubs indeed capture element sets that display high cohesion and represent the more cohesive areas of the entire implementation. The fifth hub is an exception to this in being much less cohesive than its counterparts. On visual inspection, we can see that the hub in question is closer to the center mass than the rest of the hubs. We interpret this as Eclipse AST being more coupled to the core system functionalities and thus being a less cohesive part on its own account.

We explain this phenomenon as follows. The average values are calculated for all resources required by the Eclipse system. The LCOM4 metrics indicate that the average resource is generally less cohesive than a distinguished hub. This indicates that the average resource implements more partial functionalities, while a distinguished hub implements a more complete and independent functionality. Thus, the system's coupling to a hub can be expected to be larger since all communication to access a functionality is mainly between the hub and the system. While in the case of an average resource, the communication extends to all resources that implement the complete functionality. This is apparent for hub number three. The Java Compiler is a very independent unit and its communication consists only from a handful of library objects (e.g. ASTs) to receive source code and to deliver the compilation results. While the libraries are very distinctive and unique dependencies, they are composed from several hundreds of definitions (e.g. language syntax). Since the hub

has captured the compilers functionality rather well, this is highlighted in the large coupling value.

VIII. Conclusions and Future Work

In this paper, we have demonstrated an approach to capturing information for software implementation contexts by way of utilizing cohesion and coupling aware graphs laid out using force-optimization. We applied the approach introduced in Section III to the Eclipse platform in Section IV. The received graphs were analyzed, discussed, and evaluated in Sections V, VI, and VII respectively. This section concludes the paper by discussing challenges, advantages and future work related to the presented approach.

Program visualization and the mechanics utilized in it still prove to be challenging and our approach is no exception from this. Especially, the initial setup for it remains somewhat cumbersome. The production of graphs requires access to source code, a preset AST parser, a layout engine and a visualization library. However, after the initial setup, further context queries can be served automatically. Another challenge lies in defining the query contexts. Results produced by the approach are directly dependent on the provided contexts and as such the level of expertise in defining them correlates with attained sub-graph quality.

The advantages do however outweigh the remaining challenges. Use of force-optimization gives the ability to present cohesion and coupling in a very intuitive manner. The found natural layout provides visual emphasis for structures of importance. This allows even inexperienced users to make observations regarding software modifiability. Context queries to such graphs produce a medium in which it is efficient and easy to communicate about matters that would otherwise call for rigorous analysis of program dependency structures.

Ability to visualize and explore the system should prove useful when large and obfuscated systems are explored. The approach is being integrated into our DebtFlag tool [20] in order to introduce it as part of daily development activities. This also allows us to introduce interaction capabilities like dynamic highlighting, direct source code access and metrics driven partitioning to reduce edge congestion and increase clarity. Regarding research use, we are very interested in conducting studies to see if the highlighted structures can be associated with well known architectural patterns (e.g. Model-View-Controller) and problems related to them. This could also allow their identification even from fully obfuscated sources.

References

[1] T. Ball and S. G. Eick, "Software visualization in the large," *Computer*, vol. 29, no. 4, pp. 33–43, 1996.

[2] B. Price, R. Baecker, and I. Small, "An introduction to software visualization," in *Software Visualization*, J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, Eds. Cambridge MA, MIT Press, 1998, pp. 4–26.

[3] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[4] S. R. Chidamber and C. F. Kemerer, *Towards a metrics suite for object oriented design*. ACM, 1991, vol. 26, no. 11.

[5] A. Henderson-Sellers, Z. Yang, and R. Dickinson, "The project for intercomparison of land-surface parameterization schemes," *Bulletin of the American Meteorological Society*, vol. 74, no. 7, pp. 1335–1349, 1993.

[6] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, 1994.

[7] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," in *Proceedings of the International Symposium on Applied Corporate Computing*, vol. 50, 1995, pp. 75–76.

[8] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[9] P. Caserta and O. Zendra, "Visualization of the static aspects of software: a survey," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 7, pp. 913–933, 2011.

[10] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*. IEEE, 2007, pp. 92–99.

[11] G. Langelier, H. Sahraoui, and P. Poulin, "Visualization-based analysis of quality for large-scale software systems," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 214–223.

[12] R. Koschke, "Software visualization for reverse engineering," in *Software Visualization*. Springer, 2002, pp. 138–150.

[13] Y. Hu, "Efficient, high-quality force-directed graph drawing," *Mathematica Journal*, vol. 10, no. 1, pp. 37–71, 2005.

[14] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 2003, pp. 77–ff.

[15] M. Jacomy, S. Heymann, T. Venturini, and M. Bastian, "Forceatlas2, a continuous graph layout algorithm for handy network visualization," *Medialab Center of Research 560*, 2011.

[16] T. M. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and experience*, vol. 21, no. 11, pp. 1129–1164, 1991.

[17] A. Noack, "Energy models for graph clustering." *J. Graph Algorithms Appl.*, vol. 11, no. 2, pp. 453–480, 2007.

[18] J. Barnes and P. Hut, "A hierarchical o (n log n) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[19] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Tech. Rep. 66, 1999.

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

[21] J. Holvitie, M.-J. Laakso, T. Rajala, E. Kaila, and V. Leppänen, "The role of dependency propagation in the accumulation of technical debt for software implementations," in *13th Symposium on Programming Languages and Software Tools*, k. Kiss, Ed. University of Szeged, 2013, p. 6175.

[22] S. Mäkelä and V. Leppänen, "Client-based cohesion metrics for java programs," *Science of Computer Programming*, vol. 74, no. 5, pp. 355–378, 2009.

[23] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java (TM) Language Specification, The (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[24] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," 2009. [Online]. Available: http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154

Paper V

V

Modelling Propagation of Technical Debt

Holvitie, Johannes and Licorish, Sherlock A and Leppänen, Ville (2016). In *Software Engineering and Advanced Applications (SEAA), 2016 42th Euromicro Conference on*, pages 54–58. IEEE

2016 42th Euromicro Conference on Software Engineering and Advanced Applications

Modelling Propagation of Technical Debt

Johannes Holvitie*†, Sherlock A. Licorish‡, and Ville Leppänen*†
* *TUCS*
Turku Centre for Computer Science,
Software Development Laboratory,
Turku, Finland
† *University of Turku,*
Dept. 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*—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.**

***Keywords*-technical debt propagation; software analysis;**

I. Introduction

Sub-optimalities in the software emerge due to trade-offs, oversight, or environmental changes, and they persistently affect future iterations until seen to [1]. Technical debt management pursues introducing structure and order into these sub-optimalities so as to resolve them adequately to the software development project. Prior research on technical debt has successfully introduced technical debt identification, estimation, and decision making approaches, or described how solutions from other domains can be adopted for these phases (e.g. [2], [3]). The majority of the solutions however come with preset technology or project contexts which is problematic. Indeed, Holvitie et al. [4] have noted that technical debt is capable of propagating between components that exist in different phases of the software development life-cycle, and they have further postulated that technical debt is capable of leaving its original technology context [5]. Since both the identification and estimation phases are context dependent (assessed sub-optimalities reside in predefined technology contexts like source code implemented in the Java language), 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.

II. Related Work

A. Technical Debt Propagation and its Estimation

McGregor et al. [6] hypothesized that there are two ways for technical debt to propagate within ecosystems. Firstly, the debt of a new software asset equals "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". They also noted that multiple implementation layers can diminish debt. Secondly, they establish that the user of an asset did not accumulate technical debt directly, but felt its effects indirectly. Finally, they note that compounding debt may become larger than the sum of its sources [6].

Schmid [7] provides a formal definition for technical debt accumulation. An evolution step is defined as an externally observable behavior change that introduces a characteristic to a system. Technical debt accumulation (interpretable 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.

Regarding, especially value, estimation of identified technical debt, Zazworka et al. [8] note from their case-study that principal and interest characteristics of technical debt are not bound to the type of technical debt. Eisenberg [9] notes that threshold based management approaches require defining the cost associated with reducing each type of technical debt. Falessi et al. [10] 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. also note McGegor et al.'s [6] compound property.

B. Software Entity Interconnections

Kim et al. [11] discuss an approach for classifying software changes. They first extract change history for projects from software configuration management systems. The bug-introducing changes are then identified and feature extraction is applied for them in order to produce a classifier.

2376-9505/16 $31.00 © 2016 IEEE
DOI 10.1109/SEAA.2016.53

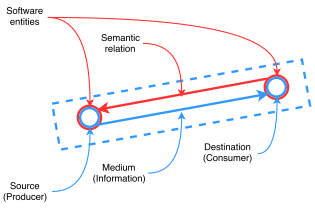


Figure 1. Software entity relationship (red; top descriptions) with a super imposed technical debt channel (blue; bottom descriptions)

Notably, bug-introducing changes are identified by backtracking from the bug-fixing change, and feature extraction takes associated log messages into account [11].

The Software Process Engineering Metamodel (SPEM) pursues formalization of software processes via definition of process components, component relations, and the impulses flowing within. Effects of the interconnections are not described by the model, but Rochd et al.'s [12] work can be seen explaining this via superimposing synchronization for the modeled components.

III. Technical Debt Propagation Channels

A. Channel Features

Our objective is to describe technical debt propagation channels capturing the effects of sub-optimal software entity alterations. The alterations correspond to software changes as argued by Holvitie et al. [5]. These changes (entity alterations) are captured here as Entity-Relationships (ER) aligned with Kagdi et al.'s [13] 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 propagation process for—i.e. the channels available to—instances of technical debt in the project specific context set).

Assume a collection of interconnected software entities (e.g. variable declarations and calls in the implementation technology's context and their descriptions in the documentation technology's context) to form a graph. The potential channels for software change is a super-set of the interconnection graph since the definition for a software change also considers assumptions (implicit channels in Section III-A1).

Figure 1 depicts one instance of a potential software change. As per the previous description of a software change between software entities, this directed relationship nay house a technical debt channel. If so, 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; further definition follows.

1) Medium: "a system with the capability of effecting or conveying something" (c.f. "medium", Merriam-Webster, 2016). In the technical debt context a medium is described 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*.

Suovuo et al. [14] have argued that the medium is either explicit or implicit. An explicit system relies on pre-existing context semantics (e.g. dependency invocation). 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 (e.g. developer's conceptualization between a component's design documentation and its implementation). Due to their unobtrusive nature, implicit channels are difficult to observe [14].

2) 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, a valid source entity 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. The source entity can not be the destination entity as the information would be consumed where produced with no outside observable effects, deviating from the definition of a software change(s) (c.f. [13]).

B. Information Properties

A technical debt instance has the following properties [15], [2]: a location, a principal, an interest, and an interest realization probability. The location property is directly related to the entities forming the sources and the destinations of the technical debt channels. The rest of the properties are related in the following.

1) Principal: A technical debt instance captures the increase in effort caused by sub-optimalities in a particular location within a software development project. The principal is the portion from the effort increase that corresponds to bringing the initial accumulation point for the difference to optimum [15], [2]. In Schmid's formalization [7] (see Section II) 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 initial accumulation point*.

2) 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 [1]. Thus, 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 initial accumulation point of the technical debt instance*.

3) Realization Probability: of a technical debt instance is the chance that further resource consumption is initiated by this debt. From the perspective of propagation channels, the realization probability is a measure of an entity-entity-relationship's existence. 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 realization probability measure indicates the chance of this system becoming a technical debt propagation channel. By the definition of principal and interest information, if 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.

IV. Technical Debt Modelling

A. Process

Technical debt channels describe systems for accumulating technical debt. Operationalizing such a system should hence dissipate technical debt. The software development life-cycle has multiple implementations of these systems (e.g. refactoring and -modelling) producing historical data. This can be used to identify technical debt dissipation, and be inverted in order to produce technical debt channels.

1) 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 [5]. The relations are captured for the entities' state sequences as pairs

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

Pair r indicates that entity e_1's state s_i has caused 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 as carried information and thus corresponds to the information realizing $(e_s, s_i) \rightarrow (e_d, s_j)$.

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 (not e.g. change propagation caused by feature addition efforts).

Finally, it is evident that technical debt can exist without related software changes. If an entity is created with principal for a new technical debt instance, no changes record alterations for this debt. Hence, arguably, identification of technical debt propagation channels requires historical data as it alone can record how the debt has realized.

3) Abstracting Channels to Models: corresponds to identifying a class T of technical debt channels t which have identical 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—referable or referring—type definition while the content is the information type. Observation level fixing ensures that the observed types adhere to these requirements.

Abstracting the model corresponds to removing all implementation specific details θ (e.g. names of specific methods) 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: $T \mapsto_\theta M$.

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: A bug from the Eclipse IDE (#73950 examined in [5]). According to the process description in Section IV-A, the first phase is observation level fixing. For the bug, we identify historical data and software entities via the bug report (c.f. https://bugs.eclipse.org/bugs/show_bug.cgi?id=73950) and the corresponding fix commit (c.f. https://git.eclipse.org/c/platform/eclipse.platform.debug.git/

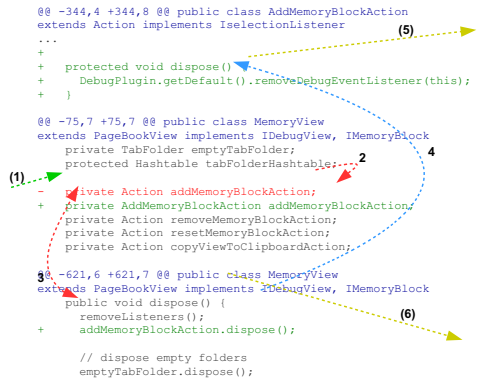


Figure 2. Transcript from an Eclipse version commit, demonstrating implicit and explicit technical debt propagation channels with directions

commit?id=9d0372b5e5159743ef53b2ec0ddaf1bfbb58a0ce). The commit describes changes at the source code level, and this allows us to observe evolution sequences e at the level of single software entities.

The second phase identifies technical debt channels. 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 bug in question, we associate the bug report's call for disposing `MemoryBlockAction` properly into changes in the fixing commit. Figure 2 is a transcript of the fixing commit. 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 (c.f. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf) 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 $source_t = type(e_s)$ is a *method invocation*. The state s_i *statement creation* is likely the first 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. Channels from one (1) to three (3) are implicit channels in Figure 2, and manual analysis is required to indicate these relationships [14]. In particular, the commit transcript cannot be solely used to decide e_s and e_d for channel three (3).

Table I. A TECHNICAL DEBT MODEL

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

The third phase of the process identifies a class of channels to abstract into a technical debt model. 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).

We may review the information properties for the captured model (see Table I). The common property for technical debt channel information 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. An argument for interest accumulation can be made for channel four (in Fig. 2), if we interpret channels one through three with their entities to precede it in the instance's propagation.

V. DISCUSSION

A. Strengths and Implications

The most important strength of the proposed approach is the accumulated library of technical debt propagation channel classes. These 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 research efforts (c.f. [15], [2]), and has potential implication for practice.

The models also expose an interface that allows programmatic evaluation of the representations; 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. Automation would arguably increase the effectiveness of technical debt management frameworks [15], [2], and pave way for more established evaluation methods [3].

Lastly, 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, to associate the model with a cost value, the historical data needs to include decomposable value information (i.e. refactoring effort).

B. Potential Challenges

Firstly, determining directions for, especially implicit, technical debt propagation channels can be difficult. As they are directed by definition, it is possible to model them from both directions (e.g. channel tree (3) in Fig. 2). Second, the identification of classes as channels is based on type libraries. Given that the amount of type defining contexts is remarkable, the amount of possible channel classes is numerous. To overcome this, arguably, a hierarchical channel taxonomy is required where the grouping dimensions exploit pre-existing taxonomies.

Third, two challenges relate to analyzing historical data 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. Examination of common change inducers could be a partial solution to this [14]. Second, the channels can only be constructed from where historical data captures technical debt dissipation. Hence, there can be channels that accumulate debt, but for which no data exist or the debt is never acted upon. The latter is arguably 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 avenues for pursuing this.

Finally, whilst two approaches [7], [6] addressed technical debt propagation, we note that neither capture the various forms and ways of technical debt propagation; focusing rather on the propagation's characteristics and capabilities. This lack of differing approaches to technical debt modelling is a challenge, as it hinders providing comparisons.

VI. Conclusion

This paper provided a theoretical description for technical debt channels as information mediums with producers and consumers. It also presented an approach for capturing technical debt channels, identifying classes of channels, and abstracting them into propagation models. In addition to advancing the technical debt research with theoretical basis for technical debt accumulation, the proposed method should deliver programmatically assessible models for automating the maintenance of manually identified technical debt information.

Future work includes exploring mechanisms for identifying taxonomies of technical debt information producers and consumers. Such mechanisms would facilitate the production of an accurate technical debt channel classification scheme. A direct application of the scheme is the identification of overlooked technical debt management areas, and indication of enhancements for existing management solutions.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

Paper VI

Mining Knowledge on Technical Debt Propagation

VI

Tomi Suovuo and Johannes Holvitie and Smed, Jouni and Leppänen, Ville (2015). In *Programming Languages and Software Tools (SPLST), 2015 Symposium on*, pages 281–295. CEUR-WS

Mining Knowledge on Technical Debt Propagation

Tomi 'bgt' Suovuo, Johannes Holvitie, Jouni Smed, and Ville Leppänen

TUCS – Turku Centre for Computer Science,
Software Development Laboratory &
University of Turku,
Department of Information Technology,
Turku, Finland
{bgt, jjholv, jouni.smed, ville.leppanen}@utu.fi

Abstract. Technical debt has gained considerable traction both in the industry and the academia due to its unique ability to distinguish asset management characteristics for problematic software project trade-offs. Management of technical debt relies on separate solutions identifying instances of technical debt, tracking the instances, and delivering information regarding the debt to relevant decision making processes. While there are several of these solutions available, due to the multiformity of software development, they are applicable only in predefined contexts that are often independent from one another. As technical debt management must consider all these aspects in unison, our work pursues connecting the software contexts via unlimited capturing and explanation of technical debt propagation intra- and inter-software-contexts. We mine software repositories (MSR) for data regarding the amount of work as a function of time. Concurrently, we gather information on events that are clearly external to the programmers' own work on these repositories. These data are then combined in an effort to statistically measure the impact of these events in the amount of work. With this data, as future work, we can apply taxonomies, code analysis, and other analyses to pinpoint these effects into different technical debt propagation channels. Abstraction of the channel patterns into rules is pursued so that development tools may automatically maintain technical debt information with them (the authors have introduced the DebtFlag tool for this). Hence, successfully implementing this study would allow further understanding and describing technical debt propagation at both the high level (longitudinal technical debt propagation effects for the project) and the low level (artifact level effects describing the mechanism of technical debt value accumulation).

1 Introduction

Technical debt is a software development concept that is interested in exposing asset management characteristics for project trade-offs [5]. Working with scarce resources to fulfill ever-changing requirements, software projects often need to

emphasize certain development driving aspects over others, such as delivery deadlines over thorough documenting. Further, invalid or lacking knowledge on certain aspects of the development may lead to emphases made that improperly reflect the actual situation. In both cases the informed and uninformed decisions result to trade-offs that accumulate technical debt [13].

It has been argued [16] that a key factor for the adoption of technical debt management into software development is the capability to produce and maintain technical debt information within the project. That is, the project trade-offs must be identified, their distribution and effects defined, and this information must be maintained to reflect the true software project state. Undoubtedly, failures in the information delivery result in unmanaged technical debt, or decisions being made based on outdated information, both of which, implicitly or explicitly, affect the project.

Technical debt research has been proficient in suggesting identification, tracking, and governance solutions to overcome the technical debt information production issues [12]. The problem is that while solutions have been proposed and trialed on various software contexts, no prior research has properly investigated the whole software context space. That is, identifying and classifying where and how technical debt exists and how does it propagate intra- and inter-software-contexts. This higher level structure may be described in some studies as the concept of technical debt interest and its accumulation, but it has not been explicitly examined; being less important to the relevant studies' goals. Arguably, however, in order to make technical debt management applicable, the various solutions must function together, and in this the enabling factor is technical debt propagation.

Today, the software projects that plug into social media services through APIs (Application Programming Interface) are an exemplar field of software context versatility. Updates to these APIs, invoked by their external authors, indicate sources of technical debt accumulation and propagation in their clients', often business critical, software. Mining Software Repositories (MSR) for the clients that are subject to these updates enables studying the software context space to address the cap in technical debt propagation knowledge.

In the 1980s software applications were relatively simple and they were delivered as is. They were relatively bug free and needed no updates. Once an application was released, any existing technical debt was outside the organization's control. As software grew increasingly complex, especially with the emergence of the Internet in the 1990s, bigger applications were released with more issues remaining. The practise eventually turned out having regularly released patches as a norm, as they were also easily distributed through the net. Technical debt was feasible and also realized. Now, in the 2010s we have complex applications that not only utilize third party libraries, but also third party services through APIs. There are regular updates to the libraries and the APIs, as well as to the client applications themselves. These all are sources of technical debt. Further, as previously shown [6], a singular technical debt instance rarely limits to a single software development component but rather spans over multiple (e.g., design,

implementation, and testing), making the emerging debt even more cumbersome to track.

Our intention is to understand the technical debt propagation context by investigating the latest trends: use of external APIs and especially those of social media services. The paper is structured as follows: we begin in Section 2 by reviewing the background. Section 3 builds on this and introduces our technical debt propagation research objectives. We introduce our approach to overcome the objectives and initial results in Section 4. The concluding remarks appear in Section 6.

2 Background

We will introduce here related work regarding technical debt, propagation in the software context, and APIs. Whilst defining core concepts for the article's foundation, empirical work is also visited so as to further understand the state of current research.

2.1 Technical Debt and Its Propagation

The term "technical debt" was initially coined by Ward Cunningham [2]. In his experience report, releasing code was paralleled to going into debt: trade-offs are made in the software project to meet a deadline, and these trade-offs can be considered debt that should be paid off when resources permit. Until the debt is paid off, it will incur interest payments—that is, later work in the project must accommodate the inoptimalities resulting from the trade-offs. This description has remained applicable to these days. Later revisits to the definition have mainly captured dimensions that further explain the role of the debt in the project: McConnell [13] provides a definition for intentional and unintentional technical debt, while Brown et al. [1] give a further description of the debt's effects via reflection to the financial domain and discussion on the resolution probability.

Firstly, McConnell [13] provided a definition for the intentionality behind the debt: intentional debt is a trade-off made whilst fully aware of its consequences, an investment with an expected return. Unintentional debt on the other hand is accumulated due to, for example, lack of knowledge. This type is a cause for concern as it remains unmanaged until discovered. Secondly, Brown et al. [1] gave a further description of the debt's effects via reflection to the financial domain: the earlier trade-offs accumulate interests payments manifesting as increased future costs, and trained decisions should evaluate if paying the interest is more profitable over reducing the loan via refactoring. Differing from the financial domain, here, the debt's interest has a probability that captures if the trade-off will have visible effects on future development: debt within a software artifact that will not be visited has a realization probability of zero.

Management of technical debt requires that we are capable of identifying and tracking the trade-offs, the atomic instances, that form the debt for a project.

Without this information readily available, trained decision regarding the debt's governance cannot be made [16]. The software context, however, makes the identification, and especially, the tracking an arduous task: instances of technical debt can span over multiple development phases and the most affected part is the software implementation [6] which arguably grows exponentially complex in the future through various abstraction layers and techniques. Nevertheless, the tracking should be able to follow a technical debt instance in this context.

From the latest systematic mapping study on technical debt [12] we can see that several solutions for tracking technical debt are available. However, we also observe (see Figure 10 in [12]) that there are areas in the software development context that are not covered by any solution; whilst most of the solutions cover sub-contexts focusing on predefined environments and specific parts of the software life-cycle. Furthermore, from Kruchten et al. [10] and Izurieta et al. [7] we can see that the causes for technical debt are various and they can be described using various characteristics. We consider all these findings indicative of the multiformity of the context of technical debt in software projects. Thus, in addition to searching for solutions in this context, technical debt research should pursue mapping the full context space and an understanding of technical debt's value in it.

Lastly, we note that technical debt tracking is the process of indicating technical debt propagation in the software context. To this end, the authors identify only the work by McGregor et al. [14] to explicitly address this issue. Here, considering mainly the software implementation, they note that technical debt for a new software asset is affected by the technical debt in relied upon assets, the amount of abstraction layers may diminish the amount of technical debt that propagates, and, in another scenario, rather than being directly accumulated from integrated assets, the technical debt has an indirect effect on the asset's users—for example, by making adoption more difficult.

2.2 Software Change Analysis

What is pursued herein is a better understanding of the context of technical debt propagation in software. We argue that *software change* should be considered the fundamental unit for this. Something that Schmid [15] also considered core to technical debt modelling during software evolution. Capturing software changes and distinguishing between technical debt inclined and other changes (that is, changes using information relatable to technical debt properties described by Brown et al. [1] and discussed in Section 2.1, and changes with no such properties) would allow non-restricted observation of technical debt in the full software context. Identifying software change retrospectively for projects corresponds to Mining Software Repositories (MSR).

Kagdi et al. [8] produce a taxonomy on MSR techniques, defining software change as *"the addition, deletion, or modification of any software artifact such that it alters, or requires amendment of, the original assumptions of the subject system."* Here, a source code change is indicated as the fundamental unit for

software evolution, but as the causes [10, 7] and the manifestations [6] for technical debt do not limit to the implementation, we adopt *software change* as the fundamental unit.

In this work, the mining efforts focus on large open-source, social-networking-enabled, repositories in order to maximally cover the diversity of software change. Tsay et al. [18] note that in GitHub handling of pull-requests is affected by social factors: highly discussed requests enjoy a lower acceptance rate, while submitters relations to—especially the manager of— the accepting project increases acceptance; this is supported also by [3]. Kalliamvakou et al. [9] survey GitHub as a MSR target. They conclude that the repository gives solid data on basic project properties, such as program language use, but synthesizing more abstract conclusions requires careful assessment. The main cause for concern here is GitHub's utilization as infrastructure for personal projects. This form of usage vastly deviates from others. To counter this bias, Kalliamvakou et al. [9] suggest considering only projects with more than two authors and demonstrated activity in both commit and pull requests.

3 Seeking Technical Debt Knowledge

In the following we address our ongoing technical debt propagation research on two distinct levels: the inter-dependency effects at the software artifact level and the longitudinal effects at the project level.

3.1 Inter-Dependency Effects within Software Artifacts

As discussed in Section 2, a multitude of solutions exist for both identifying and tracking technical debt. However, most of the solutions are intended for pre-defined software development contexts; for example, limiting their use to a specific sub-set of implementation techniques and herein, during continued software development, to certain mechanisms for technical debt propagation.

However, the ability to produce exhaustive technical debt information requires that all possibilities for technical debt propagation are acknowledged. We postulate, based on the properties of technical debt identified by Brown et al. [1] and to the average cover of single technical debt instances queried by Holvitie et al. [6], that the propagation "stream" for technical debt is capable of leaving the current host technique and merging into others. This is indicative of several sub-areas within technical debt research.

Foremost research area for technical debt propagation in software artifacts, is (1) to show that technical debt propagates between software components that can exist in external and independent projects and be implemented using different technologies. The interest and even the whole initial debt can be created in an external, but linked project that is worked by another team. The works referred here do not dispute this information, and may even implicitly assume this, but it is important to recognize this phenomenon explicitly and have quantitative research conducted on it to indisputably point it out.

Second research area, partially reliant on the first, is (2) to accumulate a documentation that describes the possible ways in which technical debt can propagate. Preferably, this would be a taxonomy capturing the unique propagation channels for technical debt. Finally, in order to enable information delivery for technical debt management purposes, (3) the channel descriptions must be enriched with information regarding technical debt value accumulation for all unique accounts of propagation. This would enable, possibly automated, technical debt information maintenance as the taxonomy is capable of tracking and valuating technical debt through out the software project.

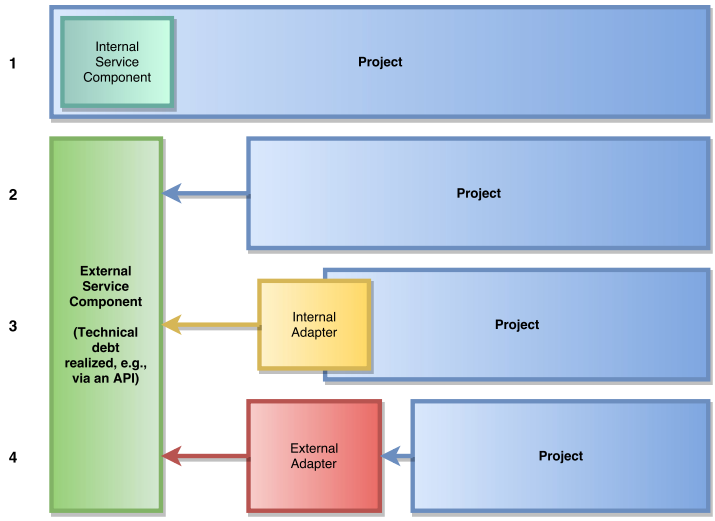


Fig. 1: Coarse classification for different *chains of projects* (COP)

3.2 The Chain of Projects

One way to identify the propagation of technical debt is to make longitudinal studies of increased debt in different phases of a project and connect them with the root causes. Technical debt can be identified as matters, such as discovered vulnerabilities, updates, and feature discontinuation in systems related to the project. Also, adding a new feature in a utilized external service API may cause technical debt when the project customer wants the new feature implemented in the project. We can identify different propagation paths by following how such

an event causes extra work in the *chain of projects* (COP) that are all linked with each other.

If an API is not interfaced directly but through a third party library, it may be that the customer is not happy to wait until the library is updated with the new feature. This will cause the project debt to be paid by implementing this new feature quickly with an internal solution. This will become a new kind of a debt, from the opposite end of the COP, when the referred library is finally updated. Here, the internal solution becomes legacy and requires refactorization into a solution that utilizes the library again, for example, in accordance to the coding conventions followed by the programming team.

There are cross waves moving back and forth in the COP from the root cause, through the library, to the end of chain application. These can be tracked by following the amount of increased work in each area.

Figure 1 demonstrates a sample classification for COPs. Here, case **1** demonstrates a monolith project that has internally implemented services with no outside dependencies. This is a classical, and probably the most studied, scenario for technical debt management, where the debt is only internally caused, felt, and managed. Cases **2** through **4** depict more modern scenarios, where the projects depend on external service providers. In case **2**, the project has a direct dependency to the service and adapts explicitly and directly as invoked by the service. A slightly dampened version, but still fully managed by the project organization is presented in case **3**, where the project, possibly alongside with the organization's other projects, uses an internally produced adapter to access the service. Hence, the project itself does not directly feel changes in the external service, but adaptation to them is still managed internally. Finally, in case **4** the project uses an external adapter to access the service. The external adapter generally serves a broader range of projects and hence is not customized for the needs of specific projects. On the other hand, external adapters tend to retain compatibility as long as possible which dampens change speeds invoked by the external service.

The classification in Figure 1 is especially important from the viewpoint of distinguishing between the "noisy" and the technical debt inclined software changes, as the monolith projects of similar size can be used as the baseline when studying how the external service invokes and propagates technical debt. Further, as per the previous description, it can be expected that the invoked technical debt will propagate quicker in the directly dependent cases than in the indirect cases **2** to **4**.

4 Exploiting Open-Source Projects

Exploiting open source code repositories enables us to make longitudinal surveys of the history. The GitHub code repository service [1] appears as a treasure trove for this kind of research. We can take a project from GitHub, and we can find for it, neatly logged, each change and its date with great detail.

[1] See `https://github.com/`

GitHub gives an open access to several different projects. However, there is also an option of hosting private projects for premium users as mentioned in Section 2.2. With only the public access to the repositories, the sample is likely to be biased. This means that traditionally non-disclosed for-profit projects cannot be found in GitHub like this, which entails that a lot of professional work is not covered by this study. However, it can be argued that functionality is delivered via the same technologies in closed-source projects.

Furthermore, regarding mapping the *software change* (as discussed in Section 2), the GitHub API gives an easy access to byte-wise size of source files and line-wise size of code change per commit. Through this we have the scale of the whole project in bytes, but the scale of changes in lines of code. Optimally both variables would be measured identically, but we can only rely on these two measures being sufficiently comparable. The only other option would be to go through the source files and count the line breaks outside the GitHub API support.

As elaborated in Section 3, we want to observe the propagation of technical debt on both at the software project and the software artifact levels, and with as little constrain as possible so as to capture the propagation context as complete as possible. Herein, we face the problem of how to identify technical debt in a highly diverse setting, and this is the reason why we emphasize the novelty of researching open-source social-networking-enabled projects.

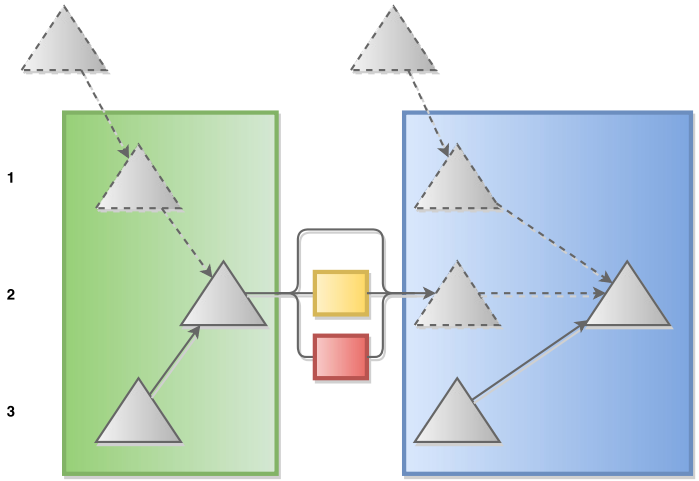


Fig. 2: Coarse classification for technical debt accumulation in projects with dependencies to external services

Figure 2 captures the different technical debt accumulation classes for projects with dependencies to external services. Case **3** depicts the most common situation in which the project accumulates technical debt that realizes at a certain point in time. In case **1** factors external to the component and its development invoke technical debt, and it may realize and invoke management needs at a point in time. In case **2** technical debt has realized (its interest probability is one, or a decision to remove the debt has been made) and it affects the project. In this scenario, the debt will propagate onwards, directly or through intermediaries, and accumulate in dependants. Accumulation channels are addressed in Figure 3.

The classification in Figure 2 is important for distinguishing technical debt inclined software change, as we must be able to distinguish between invoked change (case **2**) and internally accumulated debt (case **1** and **3**). This is because the monolith projects (see Figure 1) are able to internally accumulate technical debt, and we must form the baseline whilst aware of this.

In addition to source code, open-source projects provide access to documentation and other descriptors. Of these, the social media enabled ones form a set of projects that share a joint technical debt inducer: the social media APIs. These APIs provide business critical functionality for the projects, and every time they change, it causes several changes for their clients. Due to the massive adoption of social media services, their APIs (e.g., the Facebook Graph API [2] and the Google OpenID API [3]) integrate into and affect a vast amount of projects. This diverse collection of technologies, which all connect to the APIs that now cause changes for them, unveils a unique opportunity for technical debt research. As the changes propagate through various different technologies, they demonstrate a variety of technical debt propagation paths. Whilst our survey on to the social media involved open-source does not capture the full propagation space, particularly, propagation to business processes, it does yield a formidable library for the propagation of technical debt in delivered software and its supporting structures. Considering that usually this corresponds to the projects' delivered value, research should have a special interest to it.

Figure 3 demonstrates two channels, from a plethora of foreseeable options, through which technical debt can propagate and accumulate in new components. The upper channel captures a more problematic propagation method, in which no explicit dependency exists. In this, accumulated technical debt in the form of incomplete documentation causes a misunderstanding in a conceptualization phase of software development and leads to a complex component design. The lower channel demonstrates an explicit channel, where an interface change is felt in the dependent project as component disconnection. For example, a referred class is renamed in the service due to which the client can not access it in the original fashion. This leads to an erroneous implementation state in the dependent and undoubtedly invokes reparation efforts. In our MSR of open-source projects, over going both the human-produced messages and the automatically

[2] See `http://graph.facebook.com/`
[3] See `https://profiles.google.com/`

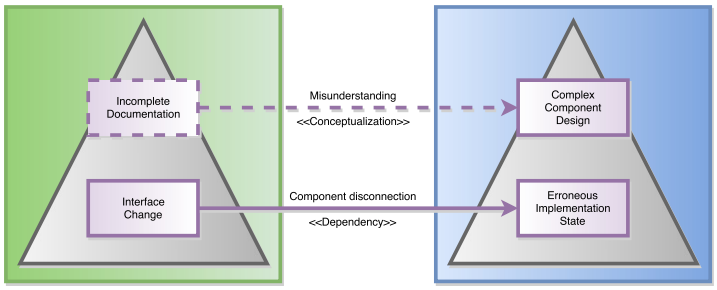


Fig. 3: Two examples of technical debt propagation channels

identified changes should reveal instances that fit both channels shown in Figure 3, but due to its implicit nature, identification of cases in the upper channel will be difficult.

4.1 Study Approach

We use the GitHub API through *PyGithub/PyGithub* library [4]. Our crawler is a Python program [5] designed to crawl through all commits of a given project and report, for each commit, the date it was committed, the amount of changes (as the amount of added and removed rows), and the changed files. As such, our crawler is in itself an end part of a COP.

For an initial test of concept we chose Google's closing of OpenID 2.0 service on April 20$^{\text{th}}$ 2015 [4] as a source of technical debt. We made a manual search in GitHub and discovered two Java projects which had closed issues mentioning Google closing the service. One was the Passport-based User Authentication system for *sails.js* applications—GitHub repository *tjwebb/sails-auth*. The other was a Grails website that provides information about festivals—GitHub repository *domurtag/festivals*. For a control project we selected another Java project that was similarly a user authentication system for *sails.js* as *sails-auth*, but did not appear to be involved with Google services—GitHub repository *waterlock/waterlock*

4.2 Initial Results

Our analysis produced the graphs shown in Figure 4.The blue colour is used for *sails-auth*, red for *festivals* and cyan for *waterlock*. The X-axis marks the time. The dots denote the amount of changes in a commit. The bars denote commits

[4] see `https://github.com/PyGithub/PyGithub` and in similar fashion for the other mentioned repositories as well
[5] GitHub repository *tomibgt/GitHubResearchDataMiner*

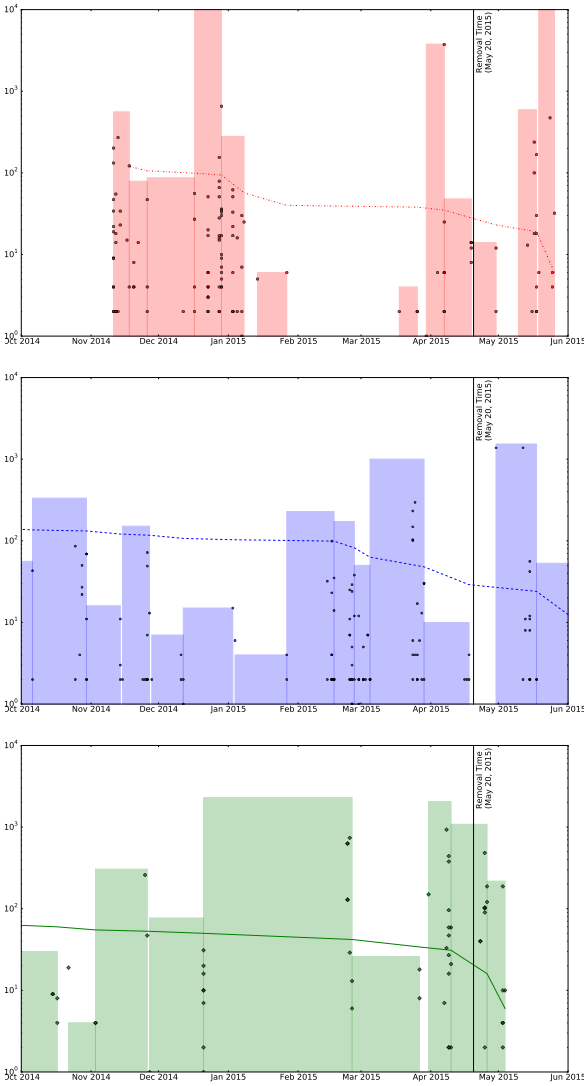


Fig. 4: Commit amount analysis for the three selected GitHub repositories

for a time period at least a week long. The lines denote commit frequency for previous time interval of at least a week. Finally, on the graph is marked the date-of-interest, April 20th 2015.

The lines show a general decline, which would appear to indicate that as a project progresses, less and less changes are made for it. Note that the Y-axis is logarithmic, which makes the lines curve down, instead of appearing linear.

It would appear to be supporting our hypothesis, where, after the marked date, *sails-auth* and *festivals* show decrease in the decline, unlike the control project *waterlock*. With only three projects and without more precise investigation we can not, of course, claim this to be strong evidence, but it is enough to encourage us in continuing with this approach.

Table 1: Commits for the *festival* repository file `show.gsp` around *Removal Time*

Time	**Add**	**Remove**	**Delta**
5/18/2015	0	2	-2
5/18/2015	7	12	-5
4/19/2015	**3**	**1**	**2**
4/19/2015	**14**	**0**	**14**
12/29/2014	11	6	5
12/29/2014	2	2	0
12/29/2014	2	1	1
12/28/2014	7	3	4
12/28/2014	8	14	-6

Table 2: Technique-wise recorded changes around *Removal Time*

Type	**Add**	**Remove**	**Delta**
`js`	86	2	84
`gsp`	35	3	32
`jpg`	.	.	.

With moderate work, the analyser can be modified to point out the files where there has been increasing changes in the commits correlating to the investigated events. (See Tables 1 and 2.) Looking into the changes made into these files should help us to analyse further the effort put by the programmers to pay the specific technical debt. Also, it should be possible to follow the wave of changes throughout the COP and analyse the propagation of the debt and the involved work and communication.

5 Applicability and Limitations

The aforedescribed approach is limited by certain factors which we would like to address here. Firstly, we described this method as a possibility to explore the complete software context space, but the study design suggests using service calls to, especially social media, APIs and libraries as the method. It can be assumed, as previously discussed, that this approach does not capture all possible varieties of *software change* (see 2.2). This is a foreseeable data limitation even though it can be argued that the volume of captured *changes* would produce a representative set for analysis; accumulating enough assurance to allow abstraction to non-captured context areas.

Second, there are limitations potentially affecting the identification of technical debt instances. We discussed the technical debt properties which can be used to associate a *software change* with managing technical debt. While this set of properties currently accounts the state-of-the-art from technical debt research, if not exhaustive, the properties may lead to missing particular sub-classes of technical debt. Approach discussed in the following paragraph, can be considered a partial remedy to this.

Finally, foreseeable limitations may also affect the tracking of technical debt instances. As a premise for tracking, [6] showed the instances' ability to span over multiple components. Modelling of the *chain of projects* was introduced as the method to allow capturing this behaviour. The current classification presented in Figure 1 considers one dimension for the COPs—presumed to be the most dominant. This classification can be a limiting factor, especially in large hybrid COP projects, but we argue that this can be countered by iteratively exploring more dimensions for the COPs until all technical debt inclined changes have been successfully associated to the technical debt instances.

Overcoming the limitations and achieving the study's objectives, there is a number of applications for the results (discussed in Section 3.1). Firstly, demonstrating technical debt's ability to propagate, almost boundlessly, between software projects and artifacts should fuel the apparent paradigm shift in software life-cycle management where the inter-connectivity of software project entities carries increased value. Second, documenting the ways in which technical debt can propagate should provide an interface for integrating knowledge from other research domains to enhance technical debt management by for example applying financial models for technical debt strategization. Lastly, associating the documentation's technical debt propagation channels with information regarding their value accumulation allows automated tooling approaches to be introduced, but also makes technical debt an integral and explicit component of the software project's value production and its assessment.

6 Conclusions and Future Work

With similar studies in the future, using different event markers, it is possible to map the propagation of technical debt by observing the amount of increased work caused by different causes of technical debt. It is possible to observe who pays the technical debt and how it is propagated from the original cause (e.g., a change in a fundamental library used by many projects) through facade libraries and components to the final applications.

In an effort to efficiently analyse the propagation of technical debt through propagation channels, a taxonomy of projects in GitHub should be created to help characterize and predict the characteristics of the projects. To this end, and to achieve the goals stated herein, we have analyzed over twenty-eight thousand projects from GitHub and have successfully identified a number of projects with references to suitable external services. According to Lambe [11], even taxonomies founded on criteria that do not stand all scrutiny, can allow for reliable

predictions and descriptions of characteristics of new members of the taxonomy based on very little information. A well created taxonomy combined with our expected mining results should help us identify different propagation channels within the projects without even analysing them at the code level. Should we find two or more clusters of different kinds of change behaviour within a single taxonomy class, it could suggest that the propagation channels between these clusters differ from each other.

There can, of course, be other causes to variance within a class. For example, it would be beneficial to have the information of the process maturity level for each project team. This kind of information would be significant in understanding the project's sensitivity to external changes and the general preparedness and carefulness in the design. [17]

Such work would provide us with a better understanding of the economy of technical debt, which again would help us give good estimates on the actual costs of applying, for example, social media APIs in an application system and compare it with the projected benefits and income. It would help in answering the question: would applying certain features increase the revenue from the service.

Acknowledgment

J. Holvitie is supported by the Nokia Foundation Scholarship and the Finnish Foundation for Technology Promotion, the Ulla Tuominen Foundation, and the Finnish Science Foundation for Economics and Technology grants.

References

1. Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., et al.: Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. pp. 47–52. ACM (2010)
2. Cunningham, W.: The WyCash portfolio management system. In: Proceedings Addendum for Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). pp. 29–30. No. 22 (1992)
3. Dabbish, L., Stuart, C., Tsay, J., Herbsleb, J.: Social coding in github: transparency and collaboration in an open software repository. In: Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work. pp. 1277–1286. ACM (2012)
4. Google Developers: Migrating to google sign-in (2015), https://developers.google.com/identity/sign-in/auth-migration
5. Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Da Silva, F., Santos, A., Siebra, C.: Tracking technical debt - an exploratory case study. In: 27th IEEE International Conference on Software Maintenance (ICSM). pp. 528–531. IEEE (2011)
6. Holvitie, J., Leppänen, V., Hyrynsalmi, S.: Technical debt and the effect of agile software development practices on it-an industry practitioner survey. In: Sixth International Workshop on Managing Technical Debt (MTD). pp. 35–42. IEEE (2014)

7. Izurieta, C., Vetrò, A., Zazworka, N., Cai, Y., Seaman, C., Shull, F.: Organizing the technical debt landscape. In: Third International Workshop on Managing Technical Debt (MTD). pp. 23–26. IEEE (2012)
8. Kagdi, H., Collard, M.L., Maletic, J.I.: 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 19(2), 77–131 (2007)
9. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 92–101. ACM (2014)
10. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: From metaphor to theory and practice. IEEE Software 29(6) (2012)
11. Lambe, P.: Organising knowledge: taxonomies, knowledge and organisational effectiveness. Chandos Publishing (2007)
12. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. Journal of Systems and Software 101, 193–220 (2015)
13. McConnell, S.: Technical debt. 10x Software Development Blog,(Nov 2007). Construx Conversations. URL= http://blogs. construx. com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx (2007)
14. McGregor, J., Monteith, J., Zhang, J.: Technical debt aggregation in ecosystems. In: Third International Workshop on Managing Technical Debt (MTD). pp. 27–30. IEEE (2012)
15. Schmid, K.: A formal approach to technical debt decision making. In: Proceedings of the 9th International ACM SIGSOFT Conference on Quality of Software Architectures. pp. 153–162. ACM (2013)
16. Seaman, C., Guo, Y., Izurieta, C., Cai, Y., Zazworka, N., Shull, F., Vetrò, A.: Using technical debt data in decision making: Potential decision approaches. In: Third International Workshop on Managing Technical Debt (MTD). pp. 45–48. IEEE (2012)
17. Suenson, E.: How Computer Programmers Work – Understanding Software Development in Practise. Ph.D. thesis, Turku Centre for Computer Science (2015)
18. Tsay, J., Dabbish, L., Herbsleb, J.: Influence of social and technical factors for evaluating contribution in github. In: Proceedings of the 36th International Conference on Software Engineering. pp. 356–366. ACM (2014)

Paper VII

Technical Debt and Agile Software Development Practices and Processes: An Industry Practitioner Survey

VII

Holvitie, Johannes and Licorish, Sherlock and Spinola, Rodrigo and Hyrynsalmi, Sami and Buchan, Jim and Mendes, Thiago and MacDonnell, Steve and Leppänen, Ville (2017). *Journal article under review*

Technical Debt and Agile Software Development Practices and Processes: An Industry Practitioner Survey

Johannes Holvitie[a,b,*], Sherlock A. Licorish[c], Rodrigo O. Spínola[d,e], Sami Hyrynsalmi[f], Stephen G. MacDonell[c,h], Thiago S. Mendes[g], Jim Buchan[h], Ville Leppänen[a,b]

[a] *Turku Centre for Computer Science, Software Development Laboratory, Turku, Finland*
[b] *University of Turku, Department of Information Technology, Turku, Finland*
[c] *University of Otago, Department of Information Science, Dunedin, Otago, New Zealand*
[d] *Salvador University, Graduate Program in Systems and Computer, Salvador, Bahia, Brazil*
[e] *Federal University of Bahia, Fraunhofer Project Center for Software and System Engineering, Salvador, Brazil*
[f] *Tampere University of Technology, Pori Campus, Pori, Finland*
[g] *Federal Institute of Bahia, Information Technology Department, Santo Amaro, Bahia, Brazil*
[h] *Auckland University of Technology, School of Engineering, Computer and Mathematical Sciences, Auckland, New Zealand*

Abstract

Context: Agile software development methods are frequently applied in volatile and resource-scarce environments that are prone to the accumulation of technical debt. While the success of these methods indicates capability to manage the debt, just how this is achieved remains unknown. In fact, some have expressed concerns about the potential of these methods to *increase* debt.
Objectives: Given the popularity and perceived success of many agile projects we sought to draw on practitioner insights and experiences in order to classify the effects of agile practices on technical debt management. We explore the breadth of practitioners' knowledge about technical debt; how technical debt is manifested in projects; and the perceived effects of common agile software development practices and processes on technical debt. In doing so, we address a research gap in technical debt knowledge and provide novel and actionable managerial recommendations.
Method: We designed, tested and executed a multi-national survey questionnaire to address our objectives, receiving 184 responses from practitioners in Brazil, Finland, and New Zealand.
Results: Our findings indicate that: 1) Practitioners are aware of technical debt, although, there was under utilization of the concept, 2) Technical debt commonly resides in legacy systems, however, concrete instances of technical debt are hard to conceptualize which makes it problematic to manage, 3) Agile practices and processes that were queried help to reduce technical debt; in particular, techniques safeguarding the structure of software implementation affect technical debt management positively.
Conclusions: While the agile practices considered here are said to have a generally positive effect on technical debt from a management perspective, matters like competing stakeholder interests cause concern.

Keywords: Technical debt, Technical debt management, Agile software development, Practitioner survey

1. Introduction

Agile software development as a collective of approaches became popular in the early 2000s and has since achieved mainstream status [1]. It provided a comparatively novel approach to software development at its inception based on the key principle of iterative-incremental delivery. This allowed the new approaches to overcome some of the critical delivery problems of their predecessors [2]. The Agile Manifesto [3], capturing the fundamentals of this development paradigm, emphasizes delivery of *"working software"*. This is core to enabling the client and the developer to align their conceptions regarding what is pursued: the client acknowledges what is achievable (i.e., increment) within a specified and predetermined time period (i.e., iteration) and certain financial constraints. After the period, the developer provides the client with the increment. This has to be working software as the client must be able to operate it in order to provide the developer with feedback regarding the functionality this software is meant to deliver.

There are, however, several alternative ways that the developer can follow to produce the working software. Ar-

*Corresponding author. Tel.: +358 2 333 8673; fax: +358 2 333 8900
Email addresses: jjholv@utu.fi (Johannes Holvitie), sherlock.licorish@otago.ac.nz (Sherlock A. Licorish), rodrigo.spinola@fpc.ufba.br (Rodrigo O. Spínola), sami.hyrynsalmi@tut.fi (Sami Hyrynsalmi), stephen.macdonell@otago.ac.nz (Stephen G. MacDonell), thiagomendes@dcc.ufba.br (Thiago S. Mendes), jim.buchan@aut.ac.nz (Jim Buchan), ville.leppanen@utu.fi (Ville Leppänen)

guably, these influence and are influenced by the developer's *definition of done* [4]. All of the alternatives produce working software for the client, but some can be, for example, more complex than others. If there are future iterations and they rely on a current complex increment, it is evident that the choices to be made can affect development: it is easier to progress software development on a less complex solution. Many agile software development methods can be seen to have built-in components, practices or processes, that account for this phenomenon. For example the Scrum process's *iteration backlog* [5] in which all items intended to be implemented in the next iteration are collected. If the developer perceives the implementation to be more or less cumbersome in the next iteration, he/she may opt to reduce or increase the number of items included in the backlog.

The discussion above implicitly describes the concept of technical debt and its management as part of software development. While as a phenomenon technical debt is not new, its conceptualization is quite recent [6]. Technical debt describes the consequences of software development actions that intentionally or unintentionally prioritize client value and/or project constraints such as delivery deadlines over more technical implementation and design considerations. These include matters like achieving and sustaining test coverage or code extensibility. Conceptually, technical debt is an analog of financial debt, with associated concepts such as levels of debt, debt accrual over time and its likely consequences, and the pressure to pay back the debt at some point in time.

Technical debt should not be equated with suboptimal software and the negative effects arising from such developments, however. There are circumstances when the decision to accrue technical debt (i.e., not pay it back) has positive cost-benefit to a project. The use of technical debt ideas to compartmentalize and characterize the deviation between current and optimal software project states can provide a mechanism for asset-management-like governance of the debt [7]. For example, a decision to not spend resources on improving a working software structure that delivers the desired functionality is reasonable if available information indicates that there is no advantage or added benefit in return for this effort.

While the consideration and application of the technical debt concept have increased exponentially in the academic context [8], to the best of the authors' knowledge several aspects of the concept's use in the software industry remain unstudied; including the contexts within which the effects of technical debt are likely to have the greatest relevance and impact. In particular, few prior studies have attempted to captured the effects of common software development practices and processes on technical debt. In the same vein, as a community we are unsure about the breadth of practitioners' knowledge about technical debt and how technical debt is manifested in their projects. The word 'effect' is used here to capture if a particular practice or process is perceived to increase or decrease the size of technical debt or the positive or negative outcomes that emerge from this debt.

Knowledge pertaining to the effects of software development practices or processes on technical debt is potentially important to practitioners since they must make decisions about the development methods they will use. Without the knowledge about the method's practices' and processes' effects, these decisions may have unintended consequences on technical debt, and ultimately, teams' performance. Based on such observations, and in keeping with our desire to gain a broad range of input to underpin our understanding, we were prompted to conduct an international survey to investigate technical depth in practice. This exploratory study sought to extend our knowledge of the depth and breadth of practitioners' knowledge about technical debt; how technical debt is manifested in projects; and the perceived effects of common agile software development practices and processes on technical debt.

This work significantly extends the authors' previous preliminary contributions. A previous publication involving some of the present authors [9] reports the design, construction, and testing of the survey instrument, in addition to its execution in Finland. The previous publication describes scenarios where technical debt is used, the media in which respondents use the concept, and their prior knowledge levels. Further, the perceived effect of common agile practices and processes is queried. The effect of continued development, the causes, and the origins are also captured for technical debt.

The current work extends this initial study to a multinational one that involves participants in Brazil and New Zealand. This allowed for delivery of several novel results. Most notably, analysis of previous knowledge, conformity to given definitions and agile technique effects now consider the respondents' development roles. We also report on technical debt's effects on several software development characteristics (e.g., perceived effect on project agility). Further, due to the increased dataset size the analysis confidence is increased, and where applicable, statistical analysis accompanies handling of the results.

The remaining sections of this article are structured as follows. Section 2 describes the background of the study, focusing on related work on technical debt and agile software development. The research approach employed is described in Section 3 with the establishment of the research questions, followed by explanations of the design and implementation of the survey study that answers them. Section 4 presents the results of the survey and the subsequent analysis of these results. Key findings, implications and future work, as well as study limitations, are discussed in Section 5. Finally, concluding remarks are provided in Section 6.

2. Background

In this section we provide the study background. Firstly, we examine the concept of technical debt, including its origins and evolution. We then provide a review and evaluation of existing relevant surveys, noting the significant characteristics for technical debt. This review and evaluation provide a working definition of technical debt used in our study. Finally, agile software development and its methods are described and evaluated to provide context for the study.

2.1. Technical Debt

The term "technical debt" was coined by Ward Cunningham [6], when he described the phenomenon of meeting a release deadline by making adaptations and concessions in a product. He also outlined how the effects felt afterwards were analogous to those associated with the incurring of financial debt. Cunningham [6] acknowledged that, most often, technical debt required payback, while the inability to manage assets could lead to a complete stand-still as the interest and effects of the adaptations (or lack thereof), become unbearable.

The definition of technical debt was later revisited on a number of occasions, usually, to generalize what Cunningham had previously described for all applicable situations whilst categorizing its characteristics. Steve McConnell's definition, which separates out intentional and unintentional accumulation of technical debt [10], has been widely adopted by academia (e.g., in [11, 12] and recognized in [8]). Building on McConnell's assessment, Brown et al. [13] provide a description of the effects of technical debt during software development by relating the concept to its financial counterpart. As it is the sole definition—to the best of the authors' knowledge—with an explicit monetary reference, it is of interest from a definition perspective when engaging practitioners (see definitions provided in the Questionnaire, see `http://soft.utu.fi/tds16/questionnaire.pdf`, Question 22).

As the previously synthesized definitions are both abstract and generic, some researchers have sought to contextualize technical debt. For instance, Alves et al. [12] provided an ontology for technical debt, explaining that 13 different types of technical debt can be distinguished usefully (e.g., design, architecture, and testing debt [14, 15]). The contextualization of technical debt in their model is apparent through the type name which indicates the software context from which the type's instances originate. Kruchten et al. have provided a Technical Debt Landscape [11] (further discussed by Izurieta et al. [16]) which captures a set of components for the emergence of technical debt. In their work, the causes are introduced by placing them relative to an axis that characterizes them as related to visibility or evolvability/maintainability (see `http://soft.utu.fi/tds16/questionnaire.pdf`, Question 31 for the authors' adaptation of this).

In going one step further, a number of studies have endeavoured to validate various contextualizations and clarifications of technical debt, often through the use of survey instruments. We review these next to situate our study in the current state of knowledge.

2.2. Technical Debt Surveys

Surveys on technical debt can be broadly partitioned into two groups: those related to characterizing technical debt, and those seeking to understand how technical debt is incurred in software development. Regarding studies characterizing technical debt, Klinger et al. [17] interviewed four software architects concerning technical debt accumulation in software projects. They noted that often the decision to incur debt was derived from the motivations of non-technical stakeholders, and hence, could be affected by competing stakeholder interests (e.g., a technically suboptimal solution was chosen due to pressing business concerns). It was nevertheless concluded that quantification of technical debt should lead to making relevant information available (so as to be visible to the project team), and thus, should enable more effective project management.

Snipes et al. [18] surveyed two change control boards to understand the decision factors for defect debt governance. They identified a set of factors that affected accumulation and reduction of technical debt by relating these to the success of chosen management strategies. Snipes et al. [19] further noted that most of the software defect management resources were spent on identification and characterization, rather than the actual removal of defects (and ultimately, technical debt).

Martini & Bosch [20] study the information needs of agile software architects and product owners with regards to architectural technical debt (ATD). They analyze quantitative and qualitative data collected from four large software development companies. Findings indicate that for architect and product owner roles the information needs are different. Notably, product owners value market attractiveness and specific customer value higher than the software architects.

Martini et al. [21] also report on a study to establish causes for the accumulation of ATD as well as its refactoring. The study is executed as a multiple-case study in five large software development companies. Based on the executed interviews a series of causes for ATD is identified and validated (e.g., uncertainty of use cases in early stages). Martini et al. present two models for ATD accumulation and evolution: crisis and phases. The Crisis Model captures a contemporary phenomenon where ATD is allowed to accumulate up until the point when adding new business value is too cumbersome, and a large refactoring must be executed. The Phases Model captures the different time periods in relation to feature research, design, and implementation, that can be identified to have differing ATD accumulation properties. Hence this model can be used to acknowledge differing ATD avoidance and refactoring opportunities.

Finally, Spinola et al. [19] compiled a list of technical debt folklore and surveyed practitioners to ascertain the extent of their agreement. Consensus was indicated for: *"technical debt often accumulates via short-term optimizations, reduction of technical debt is good for morale, non-management of technical debt results in unsustainability"*, and, *"not all technical debt is negative - which is why it should not be avoided, but rather, managed"*. The study's authors note, however, that the low number of responses ($N = 37$) limits generalizability of their results.

Concerning the investigations of technical debt in software development, Lim et al. [22] interviewed 35 practitioners regarding how technical debt manifests and how it is generally managed. In their study, Lim et al. observed that 75% of their respondents initially indicated that they were not familiar with the term "technical debt". After describing the concept, those respondents who were familiar with the concept indicated that informed decisions sometimes resulted in the team incurring technical debt, and the effects of this phenomenon were long-term. Further, management of technical debt was seen to be generally difficult, as tracking non-uniform technical debt instances was a challenging exercise. Codabux et al. [23] studied, within a large software organization, how technical debt could be characterized, the debt's effects on software development, and its management procedures. They concluded that an in-house taxonomy was perceived useful for technical debt characterization, whilst explicit measures were encouraged for management (e.g., dedicated teams and task descriptions). Codabux et al. [23] found that prioritizing the management of technical debt, particularly based on stakeholders' perception of debt severity, was ranked high among the measures for countering technical debt. These results were gathered through interviewing 28 project managers.

Ernst et al. [24] surveyed 1831 respondents from three large organizations and received 536 fully completed questionnaires. They explored whether (or not) practitioners share a common technical debt definition, if issues with architectural components are amongst the main contributors of technical debt, and if there are practices and tools readily available for technical debt management and tracking. These authors established that software practitioners and managers have a uniform understanding of technical debt. In addition, these authors found that architectural choices, especially in early stages of the software development life-cycle (as noted also by Martini et al. [21]), are a major contributor of technical debt. Further, it was revealed that there is a lack of tool support for managing architectural technical debt.

In reviewing the above pool of research, it is noted that a research gap exists. While studies have examined specific aspects of technical debt, including how technical debt accumulates in software projects, decision factors for debt governance, and how technical debt could be characterized, previous work did not report on specific, concrete instances of technical debt and their effects during software development. Similarly, while agile software development practices are said to be capable of withstanding (and mitigating) the effects of technical debt [5], previous work did not examine the effects of such practices on the occurrence and management of technical debt. Furthermore, it remains unclear if and where technical debt is likely to accumulate the most in terms of the specific phases of software development. Ernst et al. note that architectural components and early-stage decisions contribute to technical debt, however, the specific development phase, especially in relation to the other software development phases remains undetermined. As such, insights into these issues could be useful for those charged with managing technical debt. Beyond recommendations for practitioners, outcomes from such investigations would also enrich the knowledge pool (and literature base) around technical debt. We next consider the literature around agile software development, which sets the tone for our research agenda and the specific techniques that are used in this work (refer to Section 3).

2.3. Agile Software Development

The Agile Manifesto [3] captures a software development philosophy that emphasizes a context in which resources are scarce and requirements volatility is high. Given the many voices in support of agile methods, these approaches have become widely adopted and studied [25]. Agile development is implemented by several methods, and according to Dybå et al. [26], of the many flavors of agile, Extreme Programming (XP) [27] and Scrum [5] are two of the most studied agile methods. Equally, these two approaches are also considered to be the most frequently adopted in the software development industry [28, 29]. We thus now briefly examine these approaches.

The XP method is seen to implement the agile manifesto's recommendations primarily through 12 practices, which are applied throughout the software development project. For example, the *On-Site customer* practice (see the Questionnaire `http://soft.utu.fi/tds16/questionnaire.pdf`, Q14 for the practices of the Extreme Programming method) calls for the customer of the project to be always available. This practice shortens feedback time, as developers can query the customer's opinion, and resolve issues rapidly, resulting in fewer costly readjustments [27]. As a complement, the Scrum method defines processes and process artifacts. Here, customer feedback (as mentioned for XP) is largely implemented in the *Iteration Review* process (see the Questionnaire `http://soft.utu.fi/tds16/questionnaire.pdf`, Q15 for the abstracted process components of the Scrum method), which calls for concluding each development iteration with a meeting wherein the *Product Owner* serves as a customer representative and provides feedback [5]. As Scrum defines processes and XP concentrates on practices, together these methods provide an adequate representation of agile software development [30]. We have thus used these methods as a basis for inves-

tigating specific software development practices and processes in this work.

3. Research Approach

In the following three subsections (Sections 3.1, 3.2, and 3.3) we explain our research approach. We first define our research questions in Section 3.1, and describe how the study is designed in Section 3.2. Thereafter, in the final section (Section 3.3), we describe how and why the study is conducted across three countries (Brazil, Finland, and New Zealand).

3.1. Research Questions

Previous reports [23, 5] indicate that the "technical debt" metaphor has been applied by some practitioners in the software development industry, and this metaphor may also be readily understood by non-technical team members (e.g., customers, managers, and sales personnel) [24]. In fact, it has been contented that such metaphors could close the communication gap between technical and non-technical individuals and teams (including project managers) [10]. The usefulness of such knowledge would in this case of course depend on individuals' perceptions about technical debt, which may be influenced by their background. However, to the best of our knowledge, previous work has not considered this issue, especially from a role and background specific perspective.

We thus examine whether technical and non-technical stakeholders share the same understanding of the 'technical debt' metaphor, how the respondents have been exposed to it, and whether they perceive the metaphor to be useful by answering the first research question:

RQ1 (a) Are there differences in various stakeholders' perceptions of technical debt, and (b) do stakeholders' backgrounds affect such perceptions?

Previous studies reviewed above have focused on particular software development contexts and made general observations about technical debt in such settings [17, 18, 22, 23]. In one instance only the views of a few specific practitioners were captured [17], while in another change control boards were engaged [18]—both studies thus provide limited access to practitioners' opinions. While others have sought input from a larger spread of practitioners [23, 24], these members were also drawn from homogeneous contexts (i.e., single or few companies, based on company sizes, or development areas). In terms of growing our understanding of technical debt, it is pertinent to ascertain how a larger cross-section of the software development community perceives technical debt in day-to-day work. In particular, we currently lack a comprehensive picture of technical debt's drivers (e.g., reasons for its emergence as well as the influence of continued software development) and the software development phases it affects. If a broad understanding is to be pursued in these matters then we must examine a range of perspectives, from those across company, sector, scale, and country boundaries. We thus outline our second research question to characterize technical debt and to address this issue:

RQ2 (a) What are the perceived drivers of technical debt, and (b) which software development phases does technical debt affect?

The success of agile software development methods [31, 29] suggests that they are indeed capable of handling (by withstanding or possibly mitigating) emerging technical debt [5]. This implies that the methods have, at least implicit, technical debt management capabilities. That said, given that decisions made during software development are based on available knowledge, and knowledge is rarely exhaustive, the accumulation of technical debt in software projects cannot be entirely avoided [10], and in some cases nor is it desired (e.g., accumulation may be strategic to achieve faster time-to-market [32]). Agile software development methods do incorporate built-in adjustment capabilities that can be exercised according to project performance, by, for example, accepting less work in successive iterations [5]. However, these potential mitigating factors aside, previous studies have not reported practitioners' perceptions regarding the capability of agile software development methods in terms of the management of technical debt. As a method is composed from the practices and processes it introduces, there could be specific practices or processes that are responsible for the project withstanding or mitigating the effects of technical debt. Further, the possibility of finding practices and processes that work towards the opposite end, increasing technical debt and amplifying its effects, is equally possible. For instance, the drive to deliver working software after each iteration may result in many shortcuts, leading to the accrual of technical debt. Validating this proposition could be useful for the community, in terms of exploiting the identified agile practices' and processes' strengths in the management of technical debt. We thus outline our third research question.

RQ3 (a) Are practices and processes of agile methods perceived to have an effect on technical debt or its management, and (b) which practices are deemed to have the most significant effect?

In RQ3 the effect could vary, for example, practices and processes may withstand, mitigate, or amplify technical debt. Among the insights that are likely to result from answering the research questions outlined here, understandings of practitioners' perceptions regarding technical debt could be useful in informing targeted education strategies. In the same vein, insights into the drivers and software development phases that contribute the most technical debt could sharpen developers' oversight towards reducing future instances of technical debt. Further, knowing which

agile practices mitigate technical debt could provide useful pointers and inform developers' strategies in terms of which practices to use and when.

3.2. Study Design and Background

The following content describes the instrument that was used to gather the industry practitioner answers to our research questions as outlined in the previous section. Specifically, a questionnaire (see `http://soft.utu.fi/tds16/questionnaire.pdf`) was developed to enable collection of suitable data. Except for contact details, the questionnaire versions used in Brazil and New Zealand are identical. The Finnish version differs from these versions by having four one word differences (e.g., to the question of "How many concurrent projects do you have?" the Brazil and New Zealand versions have an answer option wording "None (only one)" where as the Finnish version has the wording "One" instead). Additionally, one question from the Finnish version was split into two for the Brazil and New Zealand versions, but question contents are identical between these versions. Below the three subsections describe the different parts of our instrument, and how their design was related to our objectives.

3.2.1. General Information

The first part of the survey solicits general information from respondents, relating to the responding individual as well as the organization in which they work. On the individual level, the respondent is queried about their total experience with software development and the roles they typically assume in a software development project. Experience is captured as a choice from three options: under three, three to six, or over six years, which provides a rough division into novice, intermediate, and experienced practitioners; following the classification used by Salo and Abrahamsson [33]. The role aspect covers common software project roles, and is adopted from the classification promoted by Bruegge and Dutoit [34]. Each respondent may indicate any combination of these roles, or specify new ones. The roles are intentionally described independent of specific development methodologies (e.g., Scrum defines a Scrum Master role [5] which is closest to the *Facilitation* role in the survey designed herein) as most agile practitioners are held to adapt practices and processes during their projects. Under these variations roles may be defined quite differently in contrast to pre-described methods. Additionally, capturing roles that are not directly related to specific agile methods enables us to better understand the array of approaches software organizations adopt.

At the organizational level, respondents are queried about their host company and projects. For the company, its size, number of concurrent projects, and concurrent projects per team are recorded. The respondents are then asked to focus on the project they are most closely affiliated with. For this project, its software product deliverable (i.e., is it complete or stand-alone, partial, or another type of product) is established together with the delivery target (i.e., external or internal client, in addition to the software development characteristics (i.e., transparency, predictability, efficiency, sustainability, and agility). This set of characteristics appears three times in the survey to capture respondents perceptions in relation to varying matters. The characteristics are deliberately introduced with no formal definition on the assumption that the target group of skilled practitioners is aware of these principles. We were also cautious that lengthening the instrument would discourage participants completing it fully. Finally, the respondents are queried about their project team's size, as well as both the full project and the iteration cycle times.

Subsequently, practitioners' perceptions of technical debt are mapped to their general information (both individual and organizational) in answering RQ1. Beyond answering RQ1, the data captured in the general information section are also used to explore country-specific differences.

3.2.2. Development Techniques Used

The second part of the survey questionnaire focuses on agile software development techniques, and is concerned with establishing which agile practices and processes are applied by respondents during their projects. As querying an exhaustive list of available development techniques is neither sensible nor possible, due to concerns of respondent motivation [35] and the numerous synonyms and customized techniques used, the most commonly employed agile practices and processes are used as a basis for our questions. We established in Section 2.3 that XP practices together with Scrum processes (as depicted in Figures 5a and 5b respectively) cover the underpinnings of agile software development well, in addition to being frequently used [28, 29]. Hence, respondents are presented with these options. For each option, the adoption level is recorded on a five-point Likert scale, as recommended by Alreck and Settle [35], while the adoption level descriptions are adopted from the previously referenced [33].

After capturing the applied development practices and processes we next seek the respondents' perceptions of their capability in meeting a team's software development and management needs. Finally, the questionnaire solicits details related to technical debt (considered next). This part is introduced last, as we are exploring the effects of individual practices and processes on technical debt, and we are hopeful of avoiding opinion re-adjustment [36] with respect to the capability levels queried earlier. The information collected here, and in the subsequent section, is used to answer RQ2 and RQ3.

3.2.3. Technical Debt

The final section of our multipart survey questionnaire comprises questions related to technical debt. This section is divided into two parts, the first establishes the respondent's technical debt knowledge, while the second focuses

on their recollections of a specific instance of technical debt and its effects.

Establishing each respondent's technical debt knowledge in the first sub-section is enabled by querying their prior knowledge on the matter supplemented by an optional text area to capture respondents' descriptions of the technical debt concept. Further, respondents are asked in which media, and in which development scenarios, have they used the term or seen it being used. Having established prior knowledge, respondents are then asked to read two technical debt definitions (those of McConnell [10] and Brown et al. [13], as described in Section 2.1) and to indicate if their description conforms to them. We noted that Ernst et al. [24] also baseline their respondents with the former definition.

The second subsection addresses respondents' recollections of concrete technical debt and its effects. The respondents are first asked if the software development projects they work with are affected by technical debt. The above-mentioned agile practice and process lists (noted in Section 3.2.2) are reiterated, and respondents are asked to classify the effect that each practice or process has on technical debt. The effect is recorded as a choice from a five-point Likert scale ranging from very negative to very positive. The concept is explained to respondents via examples—a practice or a process has a positive effect on technical debt if it *"can for example enhance technical debt management, lower its accumulation, or decrease its effects"*. The opposite definition is given for a negative effect.

The final part of the second subsection asks respondents to provide a description of a technical debt instance that has affected their work. Similarly to Ernst et al. [24], we ask the respondent to limit their consideration to a particular software development setting. For the technical debt instance, the project phases in which the instance resides are queried, following the phase classification of Bruegge and Dutoit [34] comprising *Requirements elicitation and analysis, Design, Implementation, and Testing.* Further, the causes underlying the prevalence of technical debt are captured as a set selection. To facilitate this selection we have converted the element list provided by Kruchten et al. in the *The Technical Debt Landscape* (see Section 2.1) [11], and we have adapted it into a set of causes; interpreted as inducers of technical debt in the landscape's elements. It should also be noted that we have also included those close areas that the landscape discusses but did not classify fully as being technical debt. This choice is made in order to be able to record if practitioners are in line with Kruchten et al.'s delimitation.

Respondents are also asked if the instance resides in a component that is considered to be, either internal or external, legacy. Finally, the dynamics of the technical debt instance are investigated by querying if continued software development affects its size and/or the magnitude of its effects. In querying the instance's dynamics, the size is left intentionally vague, and is situated immediately prior to the question regarding the effects' magnitude so as to draw attention to their separation. Our reason for taking this approach relates to the argument that technical debt may be intentionally allowed to grow in size, and its effects may only be felt at the time of realization [37].

While a combination of the answers to the questions outlined above is used to tease out answers for RQ1 and RQ3, the feedback captured via the questions in this section is used in answering RQ2.

3.3. Study Implementation and Data Collection

Our study was conducted through a web-based questionnaire. We deliberately chose this channel so as to minimize data transcription errors, while maximizing usability for the respondents [38]. Google Forms (see: `https://www.google.com/forms`) was chosen to host the survey, data collection, and pre-processing.

The questionnaire contained 37 questions, 35 of which are closed (refer to `http://soft.utu.fi/tds16/questionnaire.pdf` for a copy of the survey). The respondents could choose to define a concrete instance of technical debt by completing the optional part of the survey, but otherwise, an answer was required for all closed questions posed. The open-ended questions were used to prompt respondents for further details. (However, few respondents answered these questions, as noted below.)

From the above it is noted that our survey deviated somewhat from previous surveys (as per the studies surveyed in Section 2.2), where more open-ended questions were used to solicit practitioners' feedback. Such an approach was used to enable us to obtain a more definitive identification of technical debt effects for the pre-defined agile practices and processes, in addition to capturing information about concrete instances of technical debt in a more structured manner. That said, most questions included an open, "other", option so as to make them accept all forms of answers.

After survey construction, the questionnaire was trialled within the authors' organizations (in Brazil, Finland, and New Zealand), and the pilot target group comprised software practitioners with both agile and conventional backgrounds. Adjustments to phrasing and answer options were made to ensure consistency and clarity in terms of country-specific interpretation of questions and answer options. While only one of the target countries had English as its first language (i.e., New Zealand), trials across all territories with a single English version resulted in a low number of (mis)interpretation errors (under 3%). As such, the English version was used for all territories to enhance consistency.

For each country (Brazil, Finland, and New Zealand), a questionnaire service was set up. Access to this service was anonymous for respondents. The respondents' participation was solicited from industry-affiliated member groups, mailing lists, magazines, and research partners. A cover letter was sent to all participating organizations, explaining the objectives of the study, where the study collaborators were introduced. A privacy policy was also presented

to all respondents at the beginning of the questionnaire. Respondents from all software development backgrounds were welcomed to partake in the survey. Later sections of the paper acknowledge, for example, non-agile environments by filtering the dataset according to the respondents' backgrounds.

Respondents were allowed up to three months to complete the survey (through to May 2015), and data analysis began after the survey was taken offline. All data from the three countries were consolidated into a CSV file. Answer coding (e.g., transforming Likert scale entries to integers in order to allow for ordinal comparison) was applied where required prior to combining and analyzing the respondents' entries in order to derive answers to the previously posed research questions. Our results and analysis are presented in the following Section 4.

4. Results and Analysis

As per our study design we collated the three datasets: from Brazil, Finland, and New Zealand. The Brazil dataset comprised 62 completed questionnaires ($N_{BRA} = 62$), Finland's had $N_{FIN} = 54$, and New Zealand's $N_{NZL} = 68$. Based on the target group analysis (considering the number of individual email addresses used and company size estimates for non-direct emails) we estimate the response rate to be around 15.8% for Finland and 13.6% for New Zealand. The more obfuscated target frame unfortunately disallowed producing this figure for the Brazilian data set. Due to the unsolicited nature of the survey tool, the response rate is rather low—as indicated to be the case for other similar surveys [39]. However, the online tool allows for targeting of a much larger sample (i.e., targeting industry-affiliated member groups and mailing lists) which yields—in comparison to other surveys reviewed—a much greater total response count ($N = 184$ for this survey). Further, all gathered questionnaires were complete, and, thus, did not require us to delete any response.

We analyze these responses to answer the research questions posed in Section 3.1. General, characterizing observations are presented in Section 4.1 based on the country-wise datasets and the collated dataset. The three research questions are then answered based on the collated dataset in Sections 4.2 through 4.4 respectively.

4.1. Background Characteristics for the Datasets

For all the following results we provide a summary of these statistics in our complimentary online files (see `http://soft.utu.fi/tds16/backgrounds.pdf`) for further review. To contextualize our results we first establish the magnitude of the respondents' software development activities by querying the size and the number of concurrent projects as undertaken in their organizations. The majority of the respondents indicated having between two and ten concurrent projects (circa 50%), while a larger number of concurrent projects was also quite common. The distinct country-specific distributions were also queried to further probe the pattern noted: the Brazil distribution has two peaks in the *10 - 50* and *over 250* employees categories (circa 30% and 40% shares respectively). The three categories up to *100* employees evenly capture circa 90% of the Finnish respondents' organizations, while half of the New Zealand responses indicate an organization size of *over 250* employees. Of note is that a corresponding country-specific significant statistical difference was not observed.

The number of concurrent projects undertaken by teams and the size of respondents' teams were queried next. We note that all teams typically worked on *two to five* concurrent projects, while for Brazil and Finland the average team size is smaller than those reported for New Zealand: distributions are similar between countries and peak at the *2 - 5* projects. The majority of recorded team sizes are between two and ten persons. In their study, Ernst et al. found this category to be the second most popular while the 10 to 20 people category was the most popular.

Examining the results for development iteration length, Finland and New Zealand seem quite similar, with an average iteration length of *two to three weeks*, whereas Brazil's outcome is more variable across the length categories. Lesser differences are evident when comparing the average project lengths: all countries demonstrate a rather even distribution between the *1 - 3, 4 - 6, and over 6 months* categories (circa 30% share in each). From this, projects captured for Finland seem to be slightly shorter (peak at *1 - 3 months* category) while Brazil's are longer (peak at *over 6 months* category) and New Zealand resides in the middle of the two. We suspected that organization size might be an explaining factor here, but no significant correlation was found in statistical analysis.

Further, software delivery arrangements also seem to be related to company size, as we examine the common project deliverable and its target in this light. Tending towards smaller companies, the responses from Finland demonstrate a greater emphasis on delivering complete software products (over 80% share) to external clients (over 60% share) in comparison to both sets of respondents from Brazil and New Zealand (for both countries complete software products have a 60% share, while 40% of the delivery is done for an external client).

Examining the respondents' backgrounds, we note the following when querying respondents' years of experience working with software development related activities: country-wise deviation is almost non-existent here and almost two thirds of respondents had *over six years of experience*. This category places most of our respondents in the 'experienced' category as per Section 3.2.1. Ernst et al. [24] similarly found their respondents to have on average over six years of experience.

Wide distribution of results was observed for the number of employees working in respondents' organizations, and our respondents' average project duration indicates that the survey has covered a broad organization spec-

trum. Convergent distributions were also present in respondents' background details, for example, in years of experience. While this could be interpreted as being somewhat indicative of the global state of experience present in organizations, it should also be noted that 75% of the responses have been contributed by software practitioners who have been active, and hence, influenced by organizational cultures, for at least six years.

4.2. RQ1: Technical Debt Perceptions

In proposing RQ1 we are interested in establishing if there are differences in stakeholder perceptions of technical debt, and in understanding if stakeholders' backgrounds affect those perceptions. The following subsections provide a two-part answer to this research question. The first focuses on establishing stakeholders' prior technical debt knowledge and its closeness to the given definitions while considering the respondents' backgrounds. The second identifies the communication media through which stakeholders have been exposed to the concept of 'technical debt', if they have applied the concept in various scenarios, and whether they find the concept useful.

4.2.1. Differences in Various Stakeholders' Perceptions of Technical Debt

To form a baseline, the level of prior technical debt knowledge was queried from respondents using a five point Likert scale. Following this, a selection of technical debt definitions was displayed to the respondents with a similar Likert scale so that they could indicate how close they perceived their definition of the concept of 'technical debt' to be to those presented. Spearman's rank indicated a significant correlation ($\rho = 0.389, p < 0.001$) between respondents' assumed previous knowledge and their conformity to the definitions shown, though the results here are moderate. This meant that many practitioners who noted that they have high knowledge about technical debt, also indicated that their perception of this subject closely conforms to the shown definitions.

In drawing on the respondents' backgrounds, we found that consideration of their assumed software development role led to differences in results: the definition closeness of those occupying *Client representation, Facilitation*, and *Other* roles deviated significantly from the total population's distribution ($\chi^2(df = 4) \approx 18.92, 16.82, 152.02$ for the aforementioned roles respectively). A general overview of the results, with respondents' prior knowledge and perceived closeness to the given definitions, is depicted in Figure 1a. To enable role-wise analysis Figure 1b displays the role distributions, while the knowledge subdivisions are presented in Figures 2a and 2b.

Concentrating on prior technical debt knowledge first, we can observe a rather even distribution of attitudes from Figure 1a. Categories *well* and *adequately knowledgeable* have recorded the highest responses, over 20% share. However, a rather high proportion of respondents, circa 15%, indicate that they possess *no [prior technical debt] knowledge*.

When investigating conformity to established technical debt definitions (Figure 1a), we note that both extremes attract the respondents' answers here: the middle categories of respondents' indicating their closeness to given definitions to be *far* or *very far* both accumulate less than a 10% share of the answers. Meanwhile, both extremes of *very close* and *close* as well as *no knowledge* attract more answers. This can be due to having seen the technical debt definitions prior to this question, the respondent has a clearer picture of his/her level of technical debt knowledge and his/her conformity to the presented information, and, thus, is more confident to indicate it as one of the extremes.

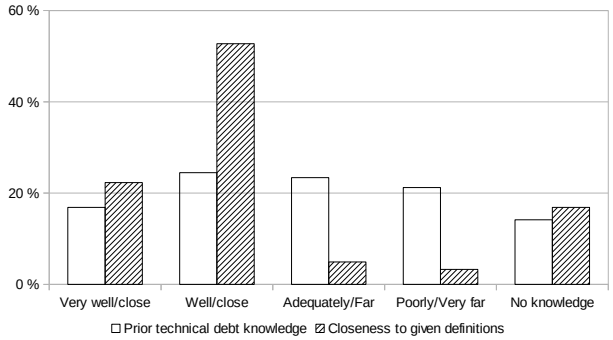
We apply the role-based subdivision to the previously presented data and present the results in Figures 2a and 2b. For these radar charts, the two highest and the two lowest answer categories have been combined so as to not clutter the charts. Looking first at prior knowledge in Figure 2a, it shows that respondents who associate with the *Design, Client Representation*, and *Facilitation* roles perceive to have the highest knowledge; however, these results were within a very small margin to the others. The lowest amount of *no knowledge* answers is indicated by those respondents in the *Design* and *Management* roles.

Role-based subdivision of indicated closeness to given definitions is given in Figure 2b. No major differences exist between roles; as was the case in the previous figure. The *Management* role seems to slightly excel the others having the highest count in the *close or very close* category and at the same time the lowest count in the *no prior knowledge* category. In examining closeness to given definitions, it should be noted that these results only indicate respondents' proximity to those features of technical debt that the particular definitions captured (discussed in Section 2.1).

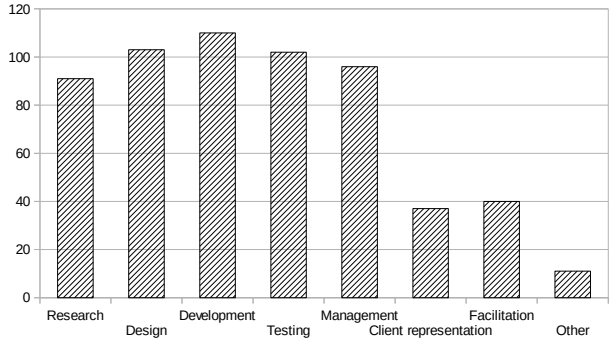
In formally evaluating our visual analysis, we conducted Spearman's rank correlation tests, relating the respondents' backgrounds to their technical debt knowledge. For assumed software development roles and conformity to given technical debt definitions, respondents associating with the *Management* role indicated closest conformity of all roles (Spearman's rho; $\rho = 0.20, p = 0.007$). Further, more experienced respondents reported both better prior technical debt knowledge ($\rho = 0.18, p = 0.013$) and closer conformity to the given definitions ($\rho = 0.18, p = 0.017$). Lastly, regarding prior knowledge on technical debt, respondents working in larger teams ($\rho = 0.21, p = 0.005$) and developing complete software systems ($\rho = 0.15, p = 0.038$) reported better prior knowledge than others.

4.2.2. Communicating The Concept of Technical Debt

In order to understand how respondents have been exposed to the 'technical debt' concept, we establish the channels through which they have encountered this term

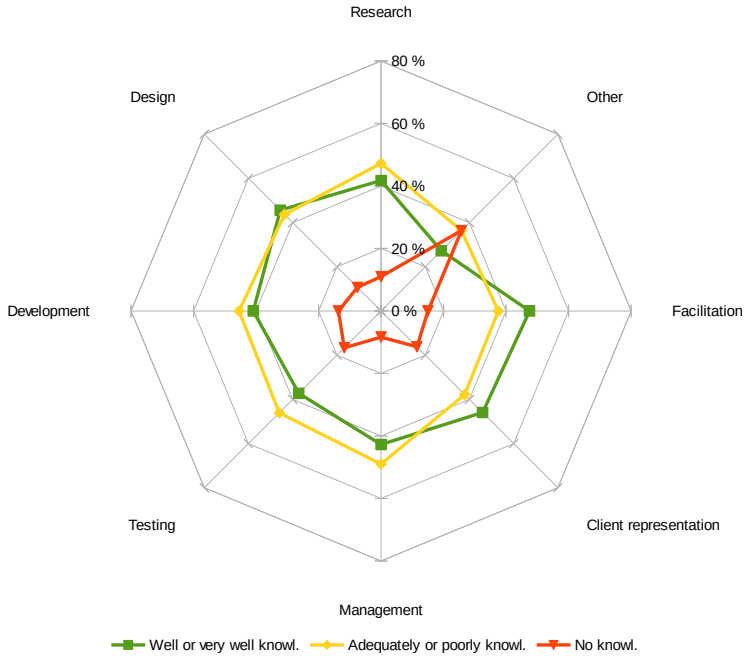


(a) Respondents' prior knowledge on technical debt and perceived conformity to presented technical debt definitions



(b) Distribution of respondents' role association. Average number of roles assumed per respondent ≈ 3.21

Figure 1: Background details for respondents



(a) Respondents' indicated prior knowledge of technical debt as a function of assumed role

(b) Respondents' claimed conformity to presented technical debt definitions as a function of assumed role

Figure 2: Role-based opinion distributions

(Figure 3a), on whose initiative, and for which reasons (Figure 3b).

Figure 3a reveals that over 40% of the respondents have either seen or heard the term used in *Work* related situations. However, *Field-specific or scientific literature* eclipses these with over 45% share. Both accounts are interesting as they seem to indicate that the concept has gained a rather solid footing in areas immediately related to contemporary software development.

News media is the least popular choice with circa 13% share of the responses. From the definition perspective, observing an increase in this share in the future could indicate that the concept has increased in popular understanding. The share of respondents who have not seen the term used is also quite high totaling 18% of the respondents.

In Figure 3b, for all provided software development related situations, 30% to 40% of the respondents indicate that the technical debt concept was used by them as well as by their colleagues. Notably, however, a circa 38% portion also indicates that the concept has not been used. In comparison, perceived usefulness is almost 25% higher than the level of current application, for all cases. This seems to indicate that the current level of applying the concept is lower than what the respondents perceive to be its potential. Finally, there is also a 15% share of respondents who perceive there to be no gains to be had from using the 'technical debt' concept.

4.3. RQ2: Technical Debt Drivers and Affected Software Development Phases

The second research question addresses the perceived drivers (causes and software development dynamics) of technical debt, and in which software development phases technical debt is seen to be more prominent. This is done by identifying and characterizing concrete instances of technical debt encountered by respondents in their software development endeavours. As per the study design, a description for a single technical debt instance was queried. From the Brazil respondents 22 ($N_{iBRA} = 22$) could provide an instance description; circa 35% of the country's total responses. The corresponding figures are $N_{iFIN} = 16$ (or c. 30%) for Finland and $N_{iNZL} = 31$ (or c. 46%) New Zealand. Noteworthy is that only a third of the respondents provided the queried description. While for many respondents, limited amount of time is a prominent explanation; argument could be made for other causes as well, namely the possible difficulty in identifying a technical debt instance for description. For the following subsections, the $N_i = 69$ descriptions (i.e., $N_{iBRA} + N_{iFIN} + N_{iNZL}$, c. 38% from all answers) form the dataset referenced. This is used to identify the drivers and affected software development phases for technical debt.

4.3.1. Drivers and Phases

Figure 3c reveals the space of possible causes for technical debt's emergence, as queried from the adopted Technical Debt Landscape (refer to Sections 2.1 and 3.2.3). In considering all responses, between four and five causes were typically cited as contributing to the emergence of a single technical debt instance. This is a notably high amount, when taking into account that the space included nine options in addition to a free description, and the causes span different areas of the software life-cycle. Cause-by-cause the dataset indicates highest rates for the *Architecture is inadequate* and the *Structure is inadequate* categories. The least frequently indicated causes of technical debt emergence are *Additional features are required* and *New features are required*. These causes are also those which the landscape's original authors noted had limited contribution to technical debt; indicative of the landscape's success in describing both technical debt and non-technical debt causes.

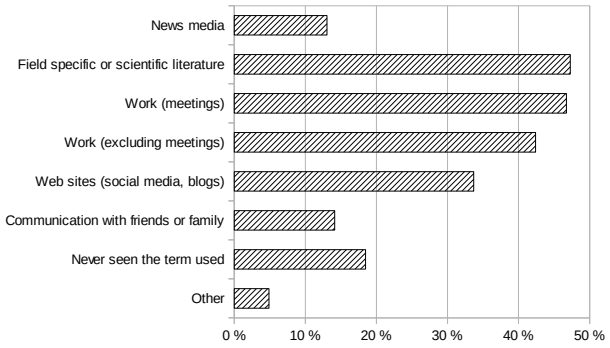
From Figure 4a, the most common origin of technical debt is indicated to be *Legacy from an earlier team/individu working on the same project/product*, with this category accounting for over 50% of the technical debt instances' origins. Around one quarter of the technical debt instances are seen not to originate from legacy components, and origins in other forms of legacy are also less frequent. However, we note a threat to validity here: these figures may be exaggerated due to respondents being more prone to selecting components that have foreign origins, and these components are often considered legacy (discussed further in Section 5.3).

As shown in Figure 4b, from the queried software development phases, in almost 90% of the cases a technical debt instance is perceived to affect the *Implementation*, and in 60% of the cases the *Design*. The design can generally be considered emergent to the implementation. Not directly visible from the figure, a single technical debt instance was seen to span between 2 and 3 software development phases in general (i.e., sum of development phases affected by an average technical debt instance is 230% in Figure 4b).

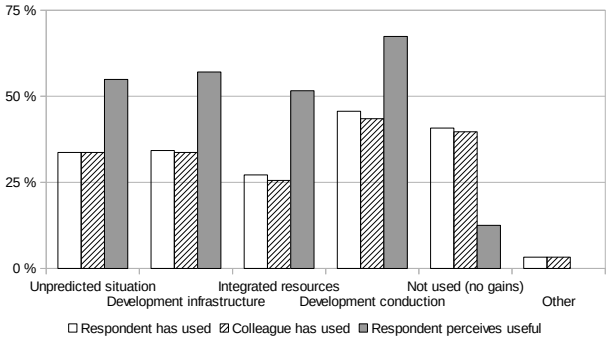
4.3.2. Dynamics of Technical Debt

In considering the views of the respondents, for circa 70% of the technical debt instances, continued software development is seen to induce an *increase* or a *large increase* in the size of the technical debt instance (Figure 4c). Martini et al. [21] also establish similar dynamics as well as constant growth inducers for ADT. However, a portion of the respondents also indicated that circa 20% of the instances experience a *decrease* or a *large decrease* in size. It is possible that in these development environments, refactoring and other technical debt management procedures are continuously executed [24, 21]. (We discussed the distinction between technical debt's size and its magnitude of effect in Section 3.2.3.)

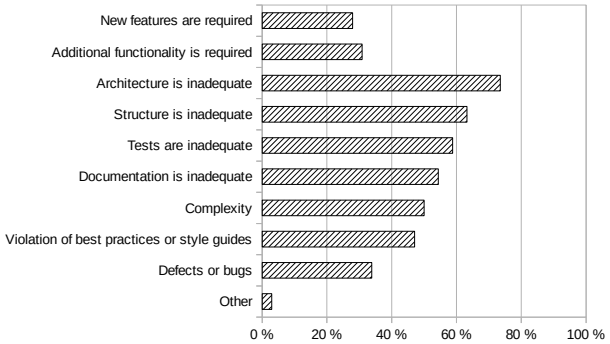
As can be seen in Figure 4d, respondents generally tend to see that the effects induced by the technical debt instance are associated with its perceived size. A 40% share of the respondents indicate the size and effects to be directly proportional, while an almost identical proportion of



(a) Media in which respondent has seen or heard the technical debt term

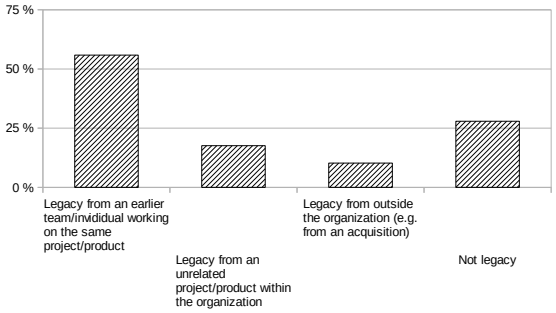


(b) Usage of the technical debt concept in software development related situations

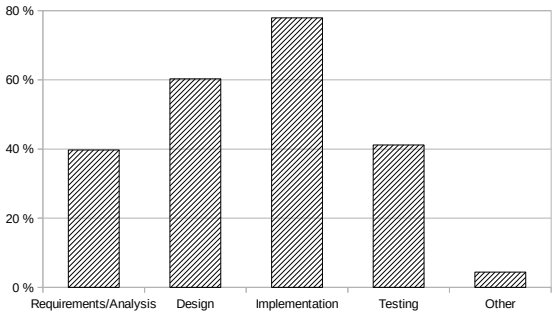


(c) Causes of concrete technical debt (based on a subset from total answers, $N_i = 69$)

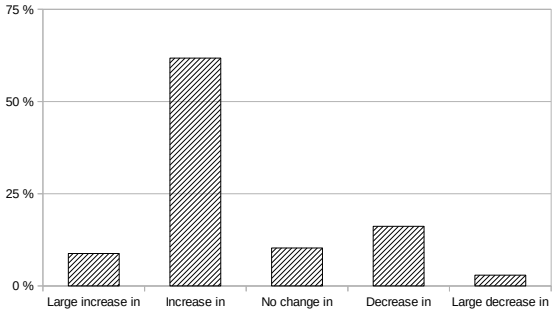
Figure 3: Technical debt usage and causes



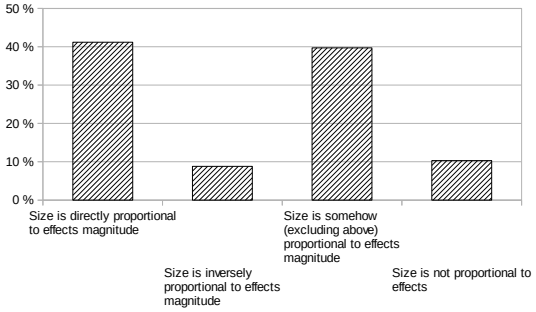
(a) Origins of concrete technical debt



(b) Software development phases affected by concrete technical debt instances (average instance spans 2 to 3 phases)



(c) Effect of continued software development on the size of concrete technical debt instances



(d) Relationship between concrete technical debt instances' sizes and the effects they induce

Figure 4: Characteristics for concrete technical debt (based on a subset from total answers, $N_i = 69$)

respondents perceive that a relationship exists but did not indicate the manner in which the technical debt instance's size alters the magnitude of its effects.

Finally, the respondents were queried if they perceived technical debt to affect the software development characteristics of their software projects. A five point Likert scale was used to capture respondent opinions ranging from significant deterioration to significant improvement for the characteristics. We performed data transformation by using an integer interval scale ranging from one to five for coding responses. In this case it was appropriate to analyze the full dataset ($N = 184$) as this question was independent of respondents' ability to define a concrete instance of technical debt. The average answer indicated technical debt as having a deteriorating effect on all queried software project characteristics (average answer fell between the options *Deteriorating effect*(2) and *No change*(3) having a standard deviation of circa one option interval).

A χ^2 test was then performed on the responses regarding the effect of technical debt on software development characteristics, and the results confirmed statistically significant relationships ($\chi^2 = 63.32, df = 20, p < 0.001$) to exist between the *predictability, efficiency, sustainability*, and *agility* characteristics. Hence, even though technical debt is seen to have a deteriorating effect on all the queried characteristics, the effect on the *transparency* characteristic fluctuates independently of the others.

4.4. RQ3: Technical Debt Management and Agile Practices

In addressing the final research question we are interested in establishing if agile practices and processes are perceived to have an effect on technical debt or its management, and if so, which practices are deemed to have the most substantial effect. To answer this question, we first establish a baseline for a specific set of agile techniques (the Scrum and XP techniques as discussed in Section 3.2.2), by surveying their level of adoption and their perceived ability to meet managerial requirements, and we then ascertain the techniques' perceived effects on technical debt. These issues are considered in the two sub-sections below.

4.4.1. Perceived Capabilities of Agile Practices

Figures 5a and 5b depict adoption levels for agile practices and processes respectively. *Coding standards, Continuous integration, 40-hour week,* and *Open office space* are the most frequently adopted practices while *Planning game, Pair-programming,* and *On-site customer* are least frequently adopted. For the queried agile processes and their artefacts no notable differences are present. All queried items average an adoption level similar to the most adopted practices from the previous group. *Iterations* are used most of then, while *Iteration reviews/retrospectives* are less frequently adopted.

We additionally report that the respondents were asked how effectively their adopted set of agile practices and processes: 1) provide coverage for the space of matters that require managing in development, and 2) meet their project's management needs. The views of the respondents from Finland and New Zealand were similar, reporting, for all practices and processes, generally adequate coverage in both dimensions. The Brazil respondents deviate from this trend: while meeting project management needs is indicated as being close to adequate for the adopted set of practices and processes, their ability to provide coverage for the space of matters that require management in development is seen to be less than adequate.

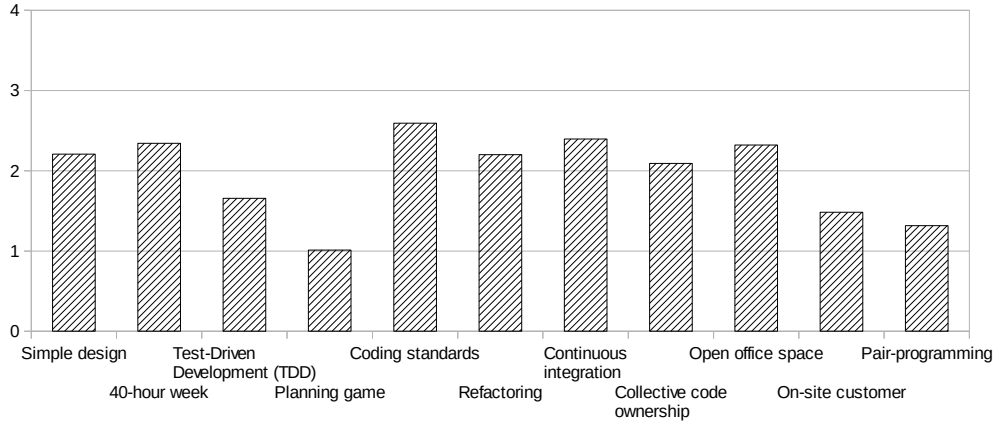
4.4.2. Effects of Agile Practices on Technical Debt

In considering the impact of agile practices and processes, Figures 6a and 6b respectively depict their perceived effects on technical debt. For both radar charts, the positive and very positive as well as the negative and very negative answer categories are combined so as to not clutter the charts. To ensure experience-based perceptions are reported, the results concentrate here only on data on adopted practices and processes (see Section 4.4.1 for discussion on adoption).

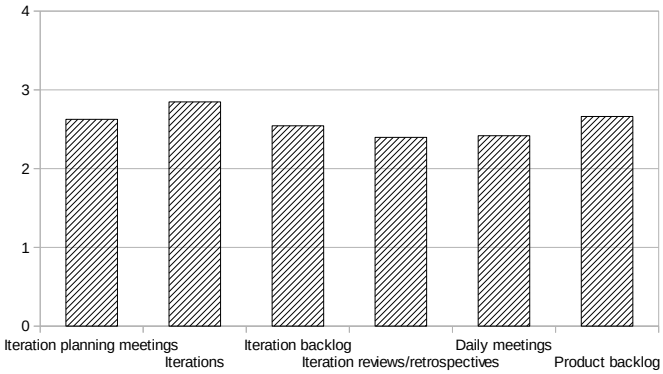
Figure 6a illustrates that, for many of the queried agile practices, the effect on technical debt is seen to be positive (c.f., Section 3.2.3). In particular, the practices of *Coding standards* and *Refactoring* are indicated as positive by over 75% of all respondents. These are followed by the *Continous Integration, Test Driven Development*, and *Collective code ownership* categories. A neutral effect is observed for the *40-hour week, Open office space*, and *Planning game* practices. Finally, the most negative effect is observed for the *On-site customer* practice. The negative share is less than 20%, but highest dispersion in effect opinions is also recorded for this practice.

The effects of agile processes and process artifacts on technical debt are much more uniform in comparison to the practices, as is evident in Figure 6b. Whilst a generally positive effect is indicated for all processes, the *Iteration reviews/retrospectives* technique enjoys the strongest positive and the lowest neutral opinion. This could be seen to support the finding of Ernst et al. [24] where 31% of respondents indicate retrospectives to be the point of identification for technical debt. The *Product backlog* process artefact seems to lead on negative effects but even in this case the share is less than 10%.

Table 1 reports, for the queried agile practices and processes, the statistically significant Spearman's rank correlations between the technique's perceived adoption level and its perceived non-negative or non-positive effect on technical debt. Both the adoption and the effect were recorded as selections from a five-point Likert scale (see Section 3.2.2). However, since the effect scale ranged from negative to positive effects, it was not directly comparable to the adoption scale. As such, the data was spliced into two for correlation testing: answers indicating non-negative effects and answers indicating non-positive effects. Finally, only answers that indicated some level of

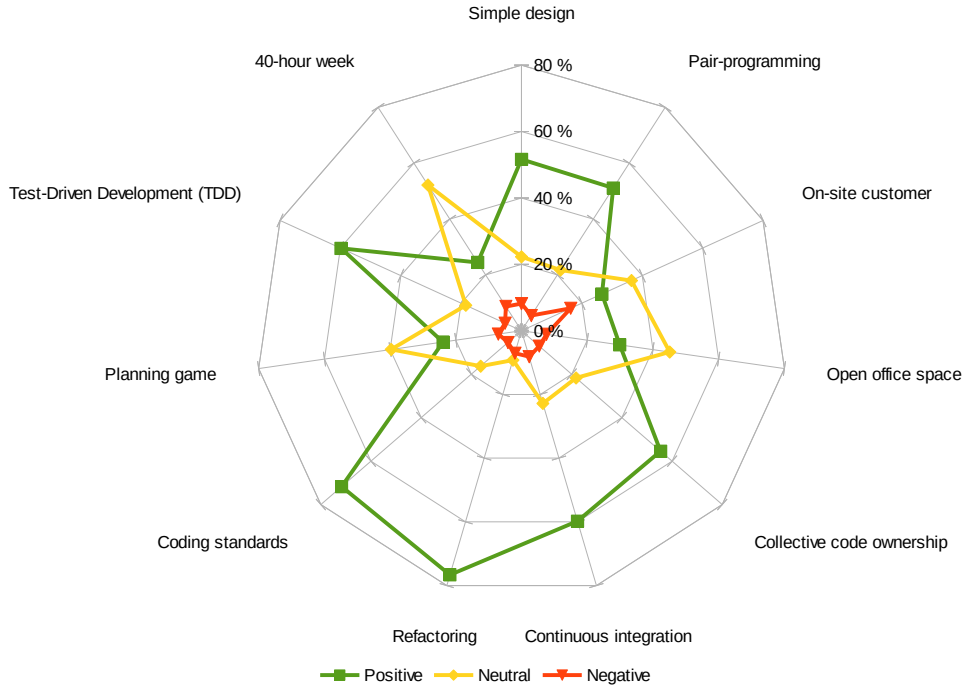


(a) For queried agile software development practices

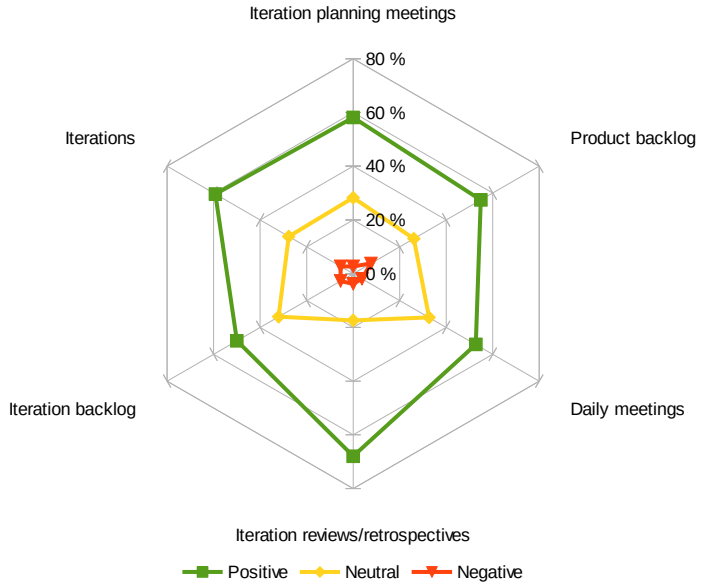


(b) For queried agile software development processes and process artefacts

Figure 5: Adoption levels from *Never used during the project* (0) to *Systematically used throughout* (4)



(a) For agile software development practices'



(b) For agile software development processes' and process artifacts'

Figure 6: Perceived effects on technical debt

Table 1: Statistically significant Spearman's rank correlations between respondents' perceived agile software development technique adoption levels and the techniques' (non-negative or non-positive) technical debt effects

	ρ	p-value	
Collective code ownership	0.231	0.009	*non-*
Iteration reviews/retrospectives	0.221	0.007	*negative*
Iteration planning meetings	0.366	0.009	*non-*
Iterations	0.415	0.001	*positive*

use for each technique were considered.

From Table 1 we note that a statistically significant correlation ($p < 0.01$) exists between the respondent indicated adoption level and the non-negative technical debt effects of the *Collective code ownership* agile practice and the *Iteration reviews/retrospectives* agile processes. Correlation here indicates that a higher perceived adoption level for a technique has lead to perceiving more positive technical debt effects for the technique, or a lower adoption level has lead to perceiving no effect for the technique. For both cases, however, the correlation can be considered weak [40].

In Table 1, a statistically significant correlation exists between the adoption level and the non-positive effects of the *Iteration planning meetings* and *Iterations* agile processes. Here the correlation is moderate and it indicates that a lower perceived adoption level for a technique has lead to perceiving more negative technical debt effects for the technique, or a higher adoption level has lead to perceiving no effect for the technique. These correlations can be considered moderate [40].

At this point it is important to note that respondents' backgrounds could be considered a threat to validity. Respondents' prior technical debt knowledge can be interpreted as indicative of their ability to acknowledge and govern technical debt within their projects. This could have an effect on projects' technical debt, independent of the applied set of development practices and processes. Thus, the effect of agile practices and processes must be observed to be independent from the respondents' prior technical debt knowledge.

Table 2 reports the statistically significant Spearman's rank correlations between respondents' perceived prior technical debt knowledge (see Figure 1a and the perceived effect of agile practices and processes on technical debt (see Figures 6a and 6b). As with Table 1 above, the data is spliced into perceived non-negative and non-positive effects prior to analysis, and only techniques that are in use are considered.

Table 2 lists those agile software development practices and processes for which statistically significant correlations (at the $p < 0.05$ level) are observed between respondents' prior technical debt knowledge and the agile techniques' indicated non-negative or non-positive technical debt effects. For the non-negative, the correlation conveys that a perceived higher technical debt knowledge has lead the respondent to indicating a more positive technical debt effect for the agile technique, or that a perceived lower technical debt knowledge has lead the respondent to indicating a neutral technical debt effect for the technique. The inversion applies for the non-positive correlations: perceived lower technical debt knowledge has lead the respondent to indicating a more negative technical debt effect for the agile technique, and a perceived higher technical debt knowledge has lead the respondent to indicating a neutral technical debt effect for the technique.

Table 2: Statistically significant Spearman's rank correlations between respondents' perceived prior technical debt knowledge and the agile software development techniques' (non-negative or non-positive) effect on technical debt

	ρ	p-value	
Simple design	0.185	0.031	*non-*
Refactoring	0.281	$3 \cdot 10^{-4}$	*negative*
Iteration planning meetings	-.170	0.032	
Iteration backlog	-.165	0.043	
Daily meetings	-.176	0.027	
40-hour week	0.201	0.034	*non-*
Coding standards	0.407	0.009	*positive*
Continuous integration	0.331	0.012	
Iteration backlog	0.261	0.030	
Daily meetings	0.269	0.028	

For all cases in Table 2 except for *Coding standards* and *Continuous integration* the correlation is weak [40]; the other correlations are moderate. The non-significant and weak correlations provide evidence for these agile practices and processes having a respondent-background-independent effect on technical debt. The moderate non-positiv effect correlations are discussed in the threats to validity (see Section 5.3).

Spearman correlation coefficients were similarly calculated for the relationship between respondents' team size and the perceived effects of agile practices and processes. Only the *Planning game practice* (Spearman's $\rho = 0.192; p = 0.032$) and the *Iteration backlog* process artefact (Spearman's $\rho = 0.226; p = 0.005$) showed significant, but weak, correlation. Thus, queried agile practice and process effects seem to be independent of team size. We further discuss these and the earlier findings in the following section

5. Discussion and Implications

This section discusses the results of our exploratory survey, first highlighting our key findings in Section 5.1. Theoretical contributions and managerial implications are next reported in Section 5.2; this section also outlines avenues for future work. Section 5.3 then considers the limitations to the work.

5.1. Key Findings

We defined three research questions for this study in Section 3.1: RQ1 (a) are there differences in various stake-

holders' perceptions of technical debt, and (b) do stakeholders' backgrounds affect such perceptions, RQ2 (a) what are the perceived drivers of technical debt, and (b) which software development phases does technical debt affect, and RQ3 (a) are practices and processes of agile methods perceived to have an effect on technical debt or its management, and (b) which practices are deemed to have the most significant effect? Effect here may be classified as withstanding, mitigating or amplifying. Sections 4.2, 4.3, and 4.4, respectively, report our results in relation to the posed research questions. In this subsection we revisit these key findings and subsequently discuss their implications for theory and practice.

First key finding: regardless of software practitioners' backgrounds we observe from their responses that they are largely aware of the concept of technical debt: their prior knowledge, in our results, was observed to be low, but the indicated closeness to the given technical debt definitions (see Figure 1a) was rather high. The findings on prior knowledge also deviate from those of Lim et al. [22] whose results indicated generally lower technical debt knowledge among their respondents. Our outcomes here suggest that awareness about—and likely also the application of—the technical debt concept is increasing amongst practitioners. In considering our outcomes regarding practitioners indicating close conformance to the provided definitions, and the definitions including descriptions for both the effects and the mechanisms related to technical debt and its accumulation, it can be argued that many professionals are equipped with knowledge that supports integration of the concept into practice. However, the practitioners are unaware of this as they assume, and thus indicated, their prior knowledge to be weaker. A possible consequence of this phenomenon is observed in Figure 3b, where current application of the technical debt concept is considerably lower (circa 35% of respondents or their colleagues applied the concept) in comparison to its perceived usefulness (circa 55%).

It should also be noted that the role-wise division of respondents' conformity to the shown definitions was uniformly high (see Figure 2b) and support the findings reported by Ernst et al. [24]. This evidence supports the premise that technical debt, at least to the extent of the discussed definitions, is a concept that can facilitate software development related discussions where all roles have mutual understanding (i.e., the concept of technical debt is understood consistently by all stakeholder groups). Hence, this could contribute towards reducing the communication gap between technical practitioners and other stakeholder groups. This potential benefit was also noted by McConnell [10].

Second key finding: technical debt is sensitive to common agile software development practices and processes (i.e., practitioners perceive these techniques to have varying effects on technical debt; see Section 4.4.2). In particular, our evidence here indicates that agile practices and processes that verify and maintain the structure and clarity of artifacts (e.g., the *Coding standards* and *Refactoring* practices) within software development projects are perceived to have a considerably positive effect from a technical debt management perspective (see Section 3.2.3 for the effect definition). We observe such a finding for the majority of the agile practices and processes in Figures 6a and 6b.

The authors perceive that the observed sensitivity is an important premise for advancing technical debt management. For the above identified agile practices and processes, the practitioner perceptions about technical debt effects need to be validated. An avenue for this is to closely track technical debt instances in environments that use these agile techniques, and to relate changes in the instances back to the techniques' use. For some techniques (e.g., the *Iteration backlog*), it is possible that the use of explicit or enhanced technical debt information (e.g., by entering technical debt instances as separate items into the backlog) in the operationalization of the technique is used as an independent variable in the validation study. Ernst et al. [24] reported findings which indicate that some technical debt management is already taking place in many of the existing processes, providing further support for choosing this validation procedure. Information produced by the validations can be used to argue for or against using particular techniques in particular software development environments.

While the generally positive results speak for integrating agile software development practices and processes for technical debt management, areas of concern still remain within these techniques. We believe that the aforementioned communication gap and especially competing stakeholder interests (discussed in Section 2.2) are potential problem areas that may result in conflicts. More particularly, the most diverse, and in our case also the most negative, effects of technical debt management were reported for an agile practice that brings together multiple roles and responsibilities (*On-site customer* in Figure 6a). This outcome supports the findings discussed earlier by Klinger et al. [17] regarding competing stakeholder interests. Competing stakeholder interests resulting in issues is interesting in the light of the first key finding. Where there is potential friction between stakeholder groups here, there is mutual understanding between roles (i.e., possible stakeholder groups) in the first key finding. To preempt this contradiction, we argue, partially based on the findings of Martini & Bosch [20] on prioritization of different development aspects, that the stakeholder groups have differing motives and as such, while possibly perceiving technical debt similarly, they can operationalize the concept (e.g., value the technical debt) differently. Additionally, we did not observe a correlation between respondents' team size (which can contribute to the number of stakeholders present) and the perceived effects of technical debt. This suggests that practitioners will not need to make provisions for this factor. The finding is interesting when we note that organization size (which may be re-

flected in team size) has been observed to affect software project complexity [41, 42] and pose new challenges, for example, for scaling agile software development methods [43].

There are undoubtedly also other aspects in addition to communication and competing stakeholder interest through which we can analyse the technical debt effects for the discussed agile techniques. In particular, time pressure and requirements volatility are aspects agile techniques try to combat against [3, 5], but projects still struggle with []. To provide an example for this alternative aspect, we can use it to explain why the *On-site customer* and *40-hour week* practices gather so diverse opinions while the *Refactoring, Continuous integration* and *Collective code ownership* practices are uniformly considered to have a highly positive technical debt effect: the *40-hour week* practice can be problematic from a scheduling perspective as it tries to control overtime work, and the *On-site customer* can represent an unwanted time pressure element and it may increase requirements volatility. At the positive end, *Refactoring* can be seen to reduce schedule volatility if it pro-actively reduces unforeseeable delays (as noted by Martini et al. [21]), *Continuous integration* minimizes lead time for completed features, and *Collective code ownership* removes bottlenecks as all stakeholder groups share the responsibility and are capable of feature delivery throughout the system.

Third key finding: industry practitioners perceive concrete technical debt to be complex and intricate which implies that it is a convoluted technical debt management subject. While only a portion of the respondents provided a concrete technical debt description (subset of total answers where $N_i = 69$), this outcome indicated that a single technical debt instance is not a quantitatively describable atomic asset that is ready for management: rather, a technical debt instance typically spans more than two software development phases (Figure 4b). In addition, a combination of several causes are responsible for technical debt emergence (Figure 3c). Furthermore, technical debt instances affect a software development project in a way that is often connected to the instances' size, but in some cases the size is not directly proportional to the effects magnitude (Figure 4c and 4d).

In light of these findings, existing and new technical debt management procedures and tools should be scrutinized to evaluate if they are truly capable of considering all the identified characteristics of concrete technical debt. For example, is a commonly used management tool or procedure capable of accommodating instances that seem to reside in and affect several software development phases at the same time? Failure on the part of the tool could mean that particular aspects of the instance remain overlooked; making the management more cumbersome. A possibility even remains that unassociated stakeholders work on the same technical debt instance unaware of one another if there is no information that ties the manifestation's of the same instance in different development phases back to the instance. Further, the convoluted nature of concrete technical debt may be part of the reason why the technical debt instances are seen to grow in size during continued development. The size increase is also concerning when we note that most technical debt instances reside in the software implementation phase. As the implementation generally corresponds to the immediate value delivered to the client, growing technical debt presence can be seen to pose negative consequences for value delivery.

We should also strive towards validating the identified features of technical debt instances through other means. For example, for those technical debt instances limited to software components that have formal semantics available, the mining software repositories [44] methods can be used to track the propagation of the instance. Examining the propagation path and finding components from several software development phases would provide supporting evidence to our finding here which saw technical debt instance to span over two software development phases.

Furthermore, for almost 75% of the technical debt instances considered by our respondents, their indicated origins are in legacy software (Figure 4a). It was noted that this figure may be biased, but the finding is still of importance. Martini et al. [21] note that legacy is a notable contributor in the accumulation of architectural technical debt, and having established this level of interplay between technical debt and legacy in the results above, we note two implications for this relationship. First, for an otherwise convoluted asset, the legacy commonality should be researched as a potential interface for technical debt management integration: the legacy characteristic does not only standardize the instances, but also calls for consulting software maintenance research for the processes used therein to identify, value, and govern legacy. Second, closeness of the 'technical debt' concept to the concept of 'software legacy' should be carefully examined to establish if professionals fully distinguish between the two concepts, and to establish that there is a need for both: technical debt pursues acknowledging asset management characteristics for particular, delimitable software project suboptimalities, while software legacy describes a general state for several project entities. However, as software legacy is seen to be a component in the majority of technical debt instances, there is a possibility that the technical debt concept is merely reduced to a synonym for legacy. In cases the newer—possibly more popular—metaphor could be used to only justify bad code whilst disregarding the investment aspects that are true to technical debt.

5.2. Theoretical Contributions and Managerial Implication

This study contributes to the ongoing academic discussion of technical debt in two respects. First, industry practitioners across multiple territories have shown high conformity (or can relate) to the given definitions for technical debt (i.e., the definitions of McConnell [10] and Brown et al. [13] queried for Figure 1a). This evidence suggests

that the technical debt concept is understood by both academic and industrial bodies. This observation provides an important premise regarding, for example, the applicability of academic solutions in industrial settings. Second, observing that respondents found the chosen selection of causes for technical debt emergence (i.e., the *The Technical Debt Landscape* queried for Figure 3c) to be suitable, both classification- and grouping-wise, for documenting instances of technical debt is indicative of its representativeness and potential utility. In addition, we observed the lowest selection frequencies for those causes that the landscape's authors have also excluded from being technical debt.

To the best of the authors' knowledge, this study is the first to verify the work of Kruchten et al. [11], and provides support for its continued development and application. Further, the finding that a predefined selection of causes captures technical debt well is somewhat divergent from the previous findings of others (e.g., [23]) where practitioners felt that a self-produced taxonomy of technical debt worked the best.

Results presented in this work may also be of use to those governing the daily activities involved in software development and the management of technical debt. First, while we observed in our results that there are differences in the level of knowledge between personnel coming from differing backgrounds, uniformity in closeness to the given definitions and in the perceived usefulness of applying the technical debt concept gives a strong indication of the capabilities of the technical debt concept as a management tool applicable for inter-stakeholder-group communication and planning.

Second, our outcomes are likely to be useful to managers in terms of the effects that common agile practices and processes have on technical debt. While our second key finding discusses the effects of the practices in more detail, we note that the results provided herein should yield valuable information to project management when evaluating the applicability of different practices and processes when used in projects. As an example of this, we note the following for agile practice: *Coding standards* was identified to have a very positive effect from a technical debt management perspective. In a recent survey [45], however, only 44% of respondents were using the practice in question. Thus, this study, among others, speaks to the importance of using coding standards; this time as one of the most important agile processes for mitigating the effects of technical debt.

We note that industry professionals, regardless of their role, should pay close attention to applying the technical debt term throughout the project life-cycle in all artifacts and processes. While country- and project-specific differences were identified in practitioners' responses, the core of the concept is in acknowledging organization-wide accumulation of technical debt that must be managed [13, 15, 46, 47]. Disregarding this fact can only reduce a project's chances of success and being maintainable in a highly competitive industry.

Finally, we anticipate three threads of research to follow on from the study presented here. First, as we have established a baseline context for technical debt and its management in the agile software development context, it would be useful to identify, classify, and describe the technical debt information that is flowing here (i.e., establishing content, producers, consumers, and effects on technical debt balance for granules of technical debt information that exist during a project life-cycle). For example, what technical debt information is associated with the most controversial *On-Site customer* practice, and how does the information change when the project advances from an initial phase (e.g., volatile requirements gathering) to a later phase (e.g., meticulous testing and release)? We expect answers to such questions to provide us with an interface through which to integrate separate, explicit, technical debt management procedures. Second, it would be useful to validate, identify, and understand the potential effect(s) of any cultural differences on the outcomes observed in this study. Such insights would help us to provide more granular recommendations for the different territories, in informing their software development practices. Such an inquiry would also benefit from larger country-specific data sets. Third, as the agile techniques, practices and processes discussed here are disconnected from their respective agile methods, we believe that there is an avenue for future work to also correlate and model the synergies between the techniques, their adoption levels, and their respective methods.

5.3. Limitations

This study provides an extension to the technical debt knowledge base, however, we concede that the work is also subject to limitations and threats to validity. The framework established by Stavru [39] allows for the systematic assessment of the study's thoroughness and trustworthiness. Thoroughness considers survey definition, design, implementation, execution, and analysis. Regarding survey definition, Sections 3.1 and 3.3 clearly describe this survey's *Objectives* and *Survey method* respectively.

For the survey's design, the *Conceptual model* was captured in discussing our early justifications and initial position in Section 3.2. The resulting *Questionnaire design* was described in this section as well. In our conceptual model we did not account for expected relationships between captured variables, as the survey is more exploratory in nature. Hence, there remains a possibility that some overlooked relationships affected the design. Section 3.3 has described the *Data collection method* whilst taking into account *Provisions for securing trustworthiness* (e.g., following established survey design principles and revisions of related work in Sections 3.2 and 2 respectively). The *Target population* for the survey is captured in Section 3.3, but, as we targeted a large sample population with a self-administrated online questionnaire, the *Sample frame* cannot be accurately described; for collection of all datasets

the available organizational registries were exhausted as described in Section 3.3. Hence, while our results indicated that we were able to contact a broad spectrum of software organizations and practitioners and we documented accurate *Sample sizes* for all the three data sets, our *Sampling method* is a *limited random sample*. However, existence of three separate datasets is likely to reduce the effects of sampling bias. We also expect further operationalization of the survey to provide additional datasets for consideration.

Regarding the survey's implementation, Section 3.3 addressed *Questionnaire evaluation* (via piloting in Section 3.3) while the *Questionnaire* itself is made accessible online at `http://soft.utu.fi/tds16/questionnaire.pdf`. For the survey's execution, the *Media* and the number of *Responses* were documented in Section 3.3. The *Response burden* was communicated to the respondents (see e.g., average response time and confidentiality guarantee in the Questionnaire (see `http://soft.utu.fi/tds16/questionnaire.pdf`), and *Execution time* was limited in Section 3.3. *Follow-up procedures* included email remainders, but had limited effect in cases where personal contact details were not available. No incentives were used to encourage response nor to counter non-response; the survey allowed respondents to give their contact information disconnected from their responses so that we could inform them about analyses' publication. Finally, in direct connection to the *Sampling frame*—and the bias discussed for it—we could only provide partial estimates for the survey *Response rate*. This poses a threat to the generalizibility of our results, and to remedy this, the work presents thorough background details analysis in Section 4.1.

Regarding the survey's analysis, its *Assessment of trustworthiness* is affected by the sample frame and response rate issues. In particular, the sample frame error is difficult to account for, given the nature of anonymous web-based surveys. Formal statistical methods are applied wherein possible to enhance trustworthiness (see Section 4), and piloting was used as an informal mechanism to reduce measurement error. *Limitations* of the analysis are addressed here via Stavru's framework [39].

Trustworthiness considers internal validity, external validity, consistency, and neutrality [48]. Regarding internal validity, in general, the survey instrument was designed so that the questions posed were straightforward (e.g., the Questionnaire `http://soft.utu.fi/tds16/questionnaire.pdf`, Q14: "Does your team apply Pair-programming?"), and hence, subject to as little misinterpretation as possible. However, issues still remain in this study that affect its internal validity. First, regarding RQ2 (see Section 3.1) and its results (see Section 4.3), we note that there is a possibility that respondents are prone to choosing software components with foreign origins for criticism. We assume this as it can be easier to pass judgment or to perceive there to be technical debt in components that you have not originated. Hence, the recorded concrete technical debt instances may be biased towards foreign origins which affects how representative our sample is. An potential negative outcome of this could be that respondents recorded a higher number of legacy origins, given that such systems are often inherited from others. Second, affecting RQ2 as well, we asked the respondents to identify and consider one technical debt instances for the duration of the survey. This means that the recorded instance was based on the subjective decisions of the respondents. This may bias the sample to, for example, describing the most prominent or cumbersome technical debt instances the respondents are facing; leaving out or limiting assessments of others. However, capturing these instances from respondents with multifold backgrounds alleviates these threats to an extent as they indicate the instances to be universally prominent to software engineering.

Section 3.2 relates the study's design to the research questions and Section 3.3 discussed piloting to verify the design (e.g., overcoming language issues discussed in Section 3.3). For external validity, we identified sample frame and response rate to affect generalizability. However, the extensive analysis of background characteristics in Section 4.1 should limit the challenges to generalization. For consistency, we have described the study design and execution in Section 3 whilst the survey instrument itself is accessible online at `http://soft.utu.fi/tds16/questionnaire.pdf`. Finally, to increase the study's neutrality we have made an effort to contrast our results against those in related works in Section 5 whilst openly discussing the study's objectives, methods, and limitations in Sections 3.1, 3.2, and 5.3 respectively.

Finally, two potential issues may affect the construct validity of this study. First, while misinterpretation of the questionnaire was likely to be reduced with unambiguous wording, it is possible that, in part, the questionnaire's terminology was difficult to comprehend. For example, discussing the principal and interest aspects of technical debt relied on the respondent's prior knowledge and comprehension of the provided technical debt definition. Misinterpretation or lack of knowledge with respect to these terms could influence how the respondent perceived the queried agile software development practices and processes to affect technical debt. Second, in Section 4.2.1 we performed statistical analysis to probe if respondents' prior technical debt knowledge correlated with what they indicated as the technical debt effect of particular agile software development practices and processes. We found from Table 2 that the correlation was moderate for *Coding standards* and *Continuous integration*. This indicates that, on these accounts, the prior knowledge of respondents is not independent from the perceived effects and thus threatens validity. However, we noted that the correlation was between prior knowledge and non-positive effects. The correlation here means that if a respondent had less prior knowledge she/he would indicate more negative effects for the technique in question or that if she/he had good prior knowledge she/he would indicate no effect for the technique. Considering this bias when interpreting Figure 6a

to answer RQ3, both the *Coding standards* and *Continuous integration* practices have a highly positive indicated effect on technical debt, which means that without the bias this effect could be even higher.

6. Conclusion

This article has reported the design and execution of a software practitioner survey in order to establish the breadth of practitioners' knowledge about technical debt, how technical debt is manifested in projects, and the perceived effects of common agile software development practices and processes on technical debt. It is concluded that while practitioners are aware of the concept of technical debt, this knowledge is implicit, and hence, underutilized (RQ1a). Noting that recent surveys [8, 24, 22] found technical debt to be a universal issue, this is a cause for concern. We observe that there were no major differences in conformance to the technical debt definitions introduced between different stakeholder groups, which indicates that technical debt can contribute to bridging the communication gap between different software stakeholder groups and roles (RQ1a). A natural next step from this result would be to look for ways in which technical debt-assisted discussions could be facilitated between the stakeholder groups. We also argue that there is a possibility of further exploring the specific role agile software development practices and processes (e.g., retrospectives) play in helping to mitigate technical debt.

We observed that instances of technical debt are convoluted in nature. Based on this, we argued that the instances would be challenging subjects for technical debt management. More closely, a single technical debt instance was seen to span multiple software development phases (RQ2b) whilst being emergent due to a number of causes (RQ2a). This poses an additional challenge for practices and tools which intend to govern these instances: a similar level of agility is required from the tool if its intention is to be able to track an instance through multiple development phases which may concern several software development components for which the stakeholders are completely independent from one another.

Notwithstanding potential bias, for the majority of the recorded technical debt instances, the indicated origins of the instances were associated with software project legacy. Further validation is required for this result, as if this relationship exists, then there is a clear motive to further look, for example, into adopting practices from the more matured software legacy domain to enhance technical debt management. Another implication we discussed from this relationship was the potential for misuse where issues previously considered legacy are re-branded to technical debt. By definition the concepts share a commonality, but, arguably, additional measures are required to convert legacy software—which is usually a large sets of software components in various states of disrepair—into manageable technical debt instances [49].

Finally, our analysis showed that most agile software development practices and processes were seen to have—eit positive or more diverged—an effect on technical debt and its management (RQ3a). Additional measures are required to identify and validate the implicit or explicit factors in these techniques that contribute towards technical debt management. Our findings here establish the interface through which technical debt management can be enhanced for existing software projects that have established development practices and process in use without introducing new and disruptive, separate technical debt management methods. To that end, techniques safeguarding the structure and state of the software implementation were perceived to have the most positive effect on technical debt (RQ3b). This is reassuring, as most technical debt instances were recorded to reside in the implementation phase of the software development life-cycle. Furthermore, our outcomes also somewhat support the competing stakeholder interest phenomenon discovered in related work as highest opinion dispersions regarding effects on technical debt were recorded for agile practices that bring together multiple roles and stakeholder groups. As several software development respondents indicated that they were applying this method, and it is indicated to be crucial for particular agile methods [27], from the perspective of technical debt management, these practices are especially interesting: there is an indicated pattern of a conflict of interests between respondents. If further studies are capable of identifying and resolving the conflict there should be an immediate positive effect on the practices' perceived technical debt effects.

Acknowledgements

Johannes Holvitie is partially supported by grants form Nokia Foundation, Ulla Tuominen Foundation, and Finnish Foundation for Technology Promotion. Dr. Rodrigo Spínola is partially supported by CNPq Universal 2014 grant 45826 /2014-9. The authors would like to thank the reviewers for their extensive comments which have significantly contributed towards the article's quality.

References

[1] B. Boehm, A view of 20th and 21st century software engineering, in: Proceedings of the 28th international conference on Software engineering, ACM, 2006, pp. 12–29.

[2] C. Larman, V. R. Basili, Iterative and incremental development: A brief history, Computer 36 (6) (2003) 47–56.

[3] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al., Manifesto for agile software development.

[4] K. Power, Definition of ready: An experience report from teams at cisco, in: International Conference on Agile Software Development, Springer, 2014, pp. 312–319.

[5] K. Schwaber, M. Beedle, Agile software development with Scrum, Vol. 18, Prentice Hall PTR Upper Saddle River^ eNJ NJ, 2002.

[6] W. Cunningham, The WyCash portfolio management system, in: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (OOPSLA), Vol. 18, 1992, pp. 29–30.

[7] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, A. Vetrò, Using technical debt data in decision making: Potential decision approaches, in: Managing Technical Debt (MTD), 2012 Third International Workshop on, IEEE, 2012, pp. 45–48.

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

[9] J. Holvitie, V. Leppanen, S. Hyrynsalmi, Technical debt and the effect of agile software development practices on it-an industry practitioner survey, in: Managing Technical Debt (MTD), 2014 Sixth International Workshop on, IEEE, 2014, pp. 35–42.

[10] S. McConnell, Technical debt, 10x Software Development Blog,(Nov 2007). Construx Conversations. URL= http://blogs. construx. com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx.

[11] P. Kruchten, R. L. Nord, I. Ozkaya, Technical debt: From metaphor to theory and practice., IEEE Software 29 (6).

[12] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, R. O. Spínola, Towards an ontology of terms on technical debt, in: Managing Technical Debt (MTD), 2014 Sixth International Workshop on, IEEE, 2014, pp. 1–7.

[13] 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.

[14] C. Seaman, Y. Guo, Measuring and monitoring technical debt, Advances in Computers 82 (2011) 25–46.

[15] E. Tom, A. Aurum, R. Vidgen, An exploration of technical debt, Journal of Systems and Software 86 (6) (2013) 1498–1516.

[16] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman, F. Shull, Organizing the technical debt landscape, in: Managing Technical Debt (MTD), 2012 Third International Workshop on, IEEE, 2012, pp. 23–26.

[17] T. Klinger, P. Tarr, P. Wagstrom, C. Williams, An enterprise perspective on technical debt, in: Proceedings of the 2nd Workshop on Managing Technical Debt, ACM, 2011, pp. 35–38.

[18] W. Snipes, B. Robinson, Y. Guo, C. Seaman, Defining the decision factors for managing defects: a technical debt perspective, in: Managing Technical Debt (MTD), 2012 Third International Workshop on, IEEE, 2012, pp. 54–60.

[19] R. O. Spinola, A. Vetro, N. Zazworka, C. Seaman, F. Shull, Investigating technical debt folklore: Shedding some light on technical debt opinion, in: Managing Technical Debt (MTD), 2013 4th International Workshop on, IEEE, 2013, pp. 1–7.

[20] A. Martini, J. Bosch, Towards prioritizing architecture technical debt: Information needs of architects and product owners (2015) 422–429.

[21] A. Martini, J. Bosch, M. Chaudron, Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study, Information and Software Technology 67 (2015) 237–253.

[22] E. Lim, N. Taksande, C. Seaman, A balancing act: what software practitioners have to say about technical debt, Software, IEEE 29 (6) (2012) 22–27.

[23] Z. Codabux, B. Williams, Managing technical debt: An industrial case study, in: Managing Technical Debt (MTD), 2013 4th International Workshop on, IEEE, 2013, pp. 8–15.

[24] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, I. Gorton, Measure it? manage it? ignore it? software practitioners and technical debt, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 50–60.

[25] T. Dingsøyr, S. Nerur, V. Balijepally, N. B. Moe, A decade of agile methodologies: Towards explaining agile software development, Journal of Systems and Software 85 (6) (2012) 1213–1221.

[26] T. Dybå, T. Dingsøyr, Empirical studies of agile software development: A systematic review, Information and software technology 50 (9) (2008) 833–859.

[27] K. Beck, C. Andres, Extreme programming explained: embrace change, Addison-Wesley Professional, 2004.

[28] N. Kurapati, V. S. C. Manyam, K. Petersen, Agile software development practice adoption survey, in: Agile processes in software engineering and extreme programming, Springer, 2012, pp. 16–30.

[29] D. West, T. Grant, Agile development: Mainstream adoption has changed agility 2 (41).

[30] P. Abrahamsson, O. Salo, J. Ronkainen, J. Warsta, Agile software development methods: Review and analysis (2002).

[31] D. Cohen, M. Lindvall, P. Costa, Agile software development, DACS SOAR Report (11).

[32] 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 (MTD), 2013 4th International Workshop on, IEEE, 2013, pp. 16–19.

[33] O. Salo, P. Abrahamsson, Agile methods in european embedded software development organisations: a survey on the actual use and usefulness of extreme programming and scrum, Software, IET 2 (1) (2008) 58–64.

[34] B. Bruegge, A. H. Dutoit, Object-Oriented Software Engineering Using UML, Patterns and Java-(Required), Prentice Hall, 2004.

[35] P. L. Alreck, R. B. Settle, The survey research handbook, Vol. 2, Irwin Homewood, IL, 1985.

[36] P. V. Marsden, J. D. Wright, Handbook of survey research, Emerald Group Publishing, 2010.

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

[38] P. M. Nardi, Doing survey research: A guide to quantitative methods.

[39] S. Stavru, A critical examination of recent industrial surveys on agile method usage, Journal of Systems and Software 94 (2014) 87–97.

[40] J. Cohen, A power primer., Psychological bulletin 112 (1) (1992) 155.

[41] C. F. Kemerer, S. Slaughter, An empirical approach to studying software evolution, Software Engineering, IEEE Transactions on 25 (4) (1999) 493–509.

[42] M. M. Lehman, Programs, life cycles, and laws of software evolution, Proceedings of the IEEE 68 (9) (1980) 1060–1076.

[43] D. J. Reifer, F. Maurer, H. Erdogmus, Scaling agile methods, Software, IEEE 20 (4) (2003) 12–14.

[44] H. Kagdi, M. L. Collard, 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 19 (2) (2007) 77–131.

[45] VersionOne, Inc., 9th annual state of agile survey, Tech. rep., available at http://www.stateofagile.com/ (2015).

[46] F. Buschmann, To pay or not to pay technical debt, Software, IEEE 28 (6) (2011) 29–31.

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

[48] E. G. Guba, Criteria for assessing the trustworthiness of naturalistic inquiries, ECTJ 29 (2) (1981) 75–91.

[49] J. Holvitie, S. A. Licorish, A. Martini, V. Leppänen, Co-existence of the technical debtand software legacyconcepts, in: 2016 1st International Workshop on Technical Debt Analytics, CEUR-WS, 2016, pp. 80–83.

Turku Centre for Computer Science
TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems.
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming – C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Images Analysis
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-Disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Market Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Frank Tétard**, Managers, Fragmentation of Working Time, and Information Systems

41. **Jan Manuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, Z_4-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development
46. **Adekunle Okunoye**, Knowledge Management and Global Diversity – A Framework to Support Organisations in Developing Countries
47. **Antonina Kloptchenko**, Text Mining Based on the Prototype Matching Method
48. **Juha Kivijärvi**, Optimization Methods for Clustering
49. **Rimvydas Rukšėnas**, Formal Development of Concurrent Components
50. **Dirk Nowotka**, Periodicity and Unbordered Factors of Words
51. **Attila Gyenesei**, Discovering Frequent Fuzzy Patterns in Relations of Quantitative Attributes
52. **Petteri Kaitovaara**, Packaging of IT Services – Conceptual and Empirical Studies
53. **Petri Rosendahl**, Niho Type Cross-Correlation Functions and Related Equations
54. **Péter Majlender**, A Normative Approach to Possibility Theory and Soft Decision Support
55. **Seppo Virtanen**, A Framework for Rapid Design and Evaluation of Protocol Processors
56. **Tomas Eklund**, The Self-Organizing Map in Financial Benchmarking
57. **Mikael Collan**, Giga-Investments: Modelling the Valuation of Very Large Industrial Real Investments
58. **Dag Björklund**, A Kernel Language for Unified Code Synthesis
59. **Shengnan Han**, Understanding User Adoption of Mobile Technology: Focusing on Physicians in Finland
60. **Irina Georgescu**, Rational Choice and Revealed Preference: A Fuzzy Approach
61. **Ping Yan**, Limit Cycles for Generalized Liénard-Type and Lotka-Volterra Systems
62. **Joonas Lehtinen**, Coding of Wavelet-Transformed Images
63. **Tommi Meskanen,** On the NTRU Cryptosystem
64. **Saeed Salehi**, Varieties of Tree Languages
65. **Jukka Arvo**, Efficient Algorithms for Hardware-Accelerated Shadow Computation
66. **Mika Hirvikorpi**, On the Tactical Level Production Planning in Flexible Manufacturing Systems
67. **Adrian Costea**, Computational Intelligence Methods for Quantitative Data Mining
68. **Cristina Seceleanu**, A Methodology for Constructing Correct Reactive Systems
69. **Luigia Petre**, Modeling with Action Systems
70. **Lu Yan**, Systematic Design of Ubiquitous Systems
71. **Mehran Gomari**, On the Generalization Ability of Bayesian Neural Networks
72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory

86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems
162. **Maryam Kamali**, Reusable Formal Architectures for Networked Systems
163. **Zhiyuan Yao**, Visual Customer Segmentation and Behavior Analysis – A SOM-Based Approach
164. **Timo Jolivet**, Combinatorics of Pisot Substitutions
165. **Rajeev Kumar Kanth**, Analysis and Life Cycle Assessment of Printed Antennas for Sustainable Wireless Systems
166. **Khalid Latif**, Design Space Exploration for MPSoC Architectures

167. **Bo Yang**, Towards Optimal Application Mapping for Energy-Efficient Many-Core Platforms
168. **Ali Hanzala Khan**, Consistency of UML Based Designs Using Ontology Reasoners
169. **Sonja Leskinen**, m-Equine: IS Support for the Horse Industry
170. **Fareed Ahmed Jokhio**, Video Transcoding in a Distributed Cloud Computing Environment
171. **Moazzam Fareed Niazi**, A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture
172. **Mari Huova**, Combinatorics on Words: New Aspects on Avoidability, Defect Effect, Equations and Palindromes
173. **Ville Timonen**, Scalable Algorithms for Height Field Illumination
174. **Henri Korvela**, Virtual Communities – A Virtual Treasure Trove for End-User Developers
175. **Kameswar Rao Vaddina**, Thermal-Aware Networked Many-Core Systems
176. **Janne Lahtiranta**, New and Emerging Challenges of the ICT-Mediated Health and Well-Being Services
177. **Irum Rauf**, Design and Validation of Stateful Composite RESTful Web Services
178. **Jari Björne**, Biomedical Event Extraction with Machine Learning
179. **Katri Haverinen**, Natural Language Processing Resources for Finnish: Corpus Development in the General and Clinical Domains
180. **Ville Salo**, Subshifts with Simple Cellular Automata
181. **Johan Ersfolk**, Scheduling Dynamic Dataflow Graphs
182. **Hongyan Liu**, On Advancing Business Intelligence in the Electricity Retail Market
183. **Adnan Ashraf**, Cost-Efficient Virtual Machine Management: Provisioning, Admission Control, and Consolidation
184. **Muhammad Nazrul Islam**, Design and Evaluation of Web Interface Signs to Improve Web Usability: A Semiotic Framework
185. **Johannes Tuikkala**, Algorithmic Techniques in Gene Expression Processing: From Imputation to Visualization
186. **Natalia Díaz Rodríguez**, Semantic and Fuzzy Modelling for Human Behaviour Recognition in Smart Spaces. A Case Study on Ambient Assisted Living
187. **Mikko Pänkäälä**, Potential and Challenges of Analog Reconfigurable Computation in Modern and Future CMOS
188. **Sami Hyrynsalmi**, Letters from the War of Ecosystems – An Analysis of Independent Software Vendors in Mobile Application Marketplaces
189. **Seppo Pulkkinen**, Efficient Optimization Algorithms for Nonlinear Data Analysis
190. **Sami Pyöttiälä**, Optimization and Measuring Techniques for Collect-and-Place Machines in Printed Circuit Board Industry
191. **Syed Mohammad Asad Hassan Jafri**, Virtual Runtime Application Partitions for Resource Management in Massively Parallel Architectures
192. **Toni Ernvall**, On Distributed Storage Codes
193. **Yuliya Prokhorova**, Rigorous Development of Safety-Critical Systems
194. **Olli Lahdenoja**, Local Binary Patterns in Focal-Plane Processing – Analysis and Applications
195. **Annika H. Holmbom**, Visual Analytics for Behavioral and Niche Market Segmentation
196. **Sergey Ostroumov**, Agent-Based Management System for Many-Core Platforms: Rigorous Design and Efficient Implementation
197. **Espen Suenson**, How Computer Programmers Work – Understanding Software Development in Practise
198. **Tuomas Poikela**, Readout Architectures for Hybrid Pixel Detector Readout Chips
199. **Bogdan Iancu**, Quantitative Refinement of Reaction-Based Biomodels
200. **Ilkka Törmä**, Structural and Computational Existence Results for Multidimensional Subshifts
201. **Sebastian Okser**, Scalable Feature Selection Applications for Genome-Wide Association Studies of Complex Diseases
202. **Fredrik Abbors**, Model-Based Testing of Software Systems: Functionality and Performance
203. **Inna Pereverzeva**, Formal Development of Resilient Distributed Systems
204. **Mikhail Barash**, Defining Contexts in Context-Free Grammars
205. **Sepinoud Azimi**, Computational Models for and from Biology: Simple Gene Assembly and Reaction Systems
206. **Petter Sandvik**, Formal Modelling for Digital Media Distribution

207. **Jongyun Moon**, Hydrogen Sensor Application of Anodic Titanium Oxide Nanostructures
208. **Simon Holmbacka**, Energy Aware Software for Many-Core Systems
209. **Charalampos Zinoviadis**, Hierarchy and Expansiveness in Two-Dimensional Subshifts of Finite Type
210. **Mika Murtojärvi**, Efficient Algorithms for Coastal Geographic Problems
211. **Sami Mäkelä**, Cohesion Metrics for Improving Software Quality
212. **Eyal Eshet**, Examining Human-Centered Design Practice in the Mobile Apps Era
213. **Jetro Vesti**, Rich Words and Balanced Words
214. **Jarkko Peltomäki**, Privileged Words and Sturmian Words
215. **Fahimeh Farahnakian**, Energy and Performance Management of Virtual Machines: Provisioning, Placement and Consolidation
216. **Diana-Elena Gratie**, Refinement of Biomodels Using Petri Nets
217. **Harri Merisaari**, Algorithmic Analysis Techniques for Molecular Imaging
218. **Stefan Grönroos**, Efficient and Low-Cost Software Defined Radio on Commodity Hardware
219. **Noora Nieminen**, Garbling Schemes and Applications
220. **Ville Taajamaa**, O-CDIO: Engineering Education Framework with Embedded Design Thinking Methods
221. **Johannes Holvitie**, Technical Debt in Software Development – Examining Premises and Overcoming Implementation for Efficient Management

Turku
Centre *for*
Computer
Science

http://www. tucs.fi

tucs@abo.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Future Technologies
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Faculty of Science and Engineering
- Computer Engineering
- Computer Science

Faculty of Social Sciences, Business and Economics
- Information Systems

ISBN 978-952-12-3524-5
ISSN 1239-1883

Johannes Holvitie

Johannes Holvitie

Johannes Holvitie

Technical Debt in Software Development

Technical Debt in Software Development

Technical Debt in Software Development