The Journal of Systems and Software 86 (2013) 1498–1516

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

An exploration of technical debt

Edith Tom[a], Aybüke Aurum[a,*], Richard Vidgen[a,b]

[a] School of Information Systems, Technology and Management, University of New South Wales, Sydney, Australia
[b] Department of Management Systems, Hull University Business School, Hull, United Kingdom

ARTICLE INFO

Article history:
Received 1 March 2012
Received in revised form
31 December 2012
Accepted 31 December 2012
Available online 21 January 2013

Keywords:
Technical debt
Code debt
Precedents
Outcomes
Benefits and drawbacks
Multivocal literature review

ABSTRACT

Context: Whilst technical debt is considered to be detrimental to the long term success of software development, it appears to be poorly understood in academic literature. The absence of a clear definition and model for technical debt exacerbates the challenge of its identification and adequate management, thus preventing the realisation of technical debt's utility as a conceptual and technical communication device.
Objective: To make a critical examination of technical debt and consolidate understanding of the nature of technical debt and its implications for software development.
Method: An exploratory case study technique that involves multivocal literature review, supplemented by interviews with software practitioners and academics to establish the boundaries of the technical debt phenomenon.
Result: A key outcome of this research is the creation of a theoretical framework that provides a holistic view of technical debt comprising a set of technical debts dimensions, attributes, precedents and outcomes, as well as the phenomenon itself and a taxonomy that describes and encompasses different forms of the technical debt phenomenon.
Conclusion: The proposed framework provides a useful approach to understanding the overall phenomenon of technical debt for practical purposes. Future research should incorporate empirical studies to validate heuristics and techniques that will assist practitioners in their management of technical debt.

1. Introduction

Technical debt is recognised as a critical issue in the software development industry: the global technical debt bill is estimated by Gartner to be $US500 billion in 2010 with the potential to double in five years' time (Thibodeau, 2011; Szykarski, 2012). Technical debt utilises financial debt (Allman, 2012; Zazworka, 2011; Wiklund et al., 2012) as a metaphor to describe the phenomenon of increasing software development costs over time. Whilst this phenomenon is evidently detrimental, e.g. increased cost, low product quality, slowed progress to the long term success of software development (Lim et al., 2012), it appears to be poorly understood in academic literature. The absence of a clear definition and model for technical debt exacerbates the challenge of its identification and adequate management. It further prevents the realisation of technical debt's utility as a conceptual and technical communication device. Whilst technical debt is readily discussed in industry, there is an evident knowledge gap between the academic and practitioner communities concerning the nature of technical debt. This increases the risk that "*intuitively attractive but sub-optimal heuristics*" (Brown et al., 2010a,b) may be adopted out of necessity by practitioners.

There is, therefore, a need to rigorously define and validate the technical debt concept so that its impacts can be understood, within the practitioner community as well as in academia. As a preliminary study we developed a systematic literature review (SLR) to understand the state of the academic research addressing technical debt (Tom et al., 2012). The study found a lack of primary studies that examined the phenomenon of technical debt. Further, there was an absence of a comprehensive definition, conceptual model, or substantive theoretical framework of technical debt in academic literature. This lack prevents an adequate understanding of the phenomenon, and presents a barrier to further research – particularly the possibility of empirical study.

The objective of this research is to consolidate understanding of the nature of technical debt and its implications for software development, thus establishing the boundaries of the phenomenon and a more complete theoretical framework to facilitate future research. The questions to be answered in this research are:

- RQ1 – What are the dimensions of technical debt?
- RQ2 – How does technical debt arise?
- RQ3 – What are the benefits and drawbacks of allowing technical debt to accrue?

* Corresponding author. Tel.: +61 293854418; fax: +61 296624061.
E-mail address: aybuke@unsw.edu.au (A. Aurum).

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.12.052

This research applies an exploratory case study technique that involves a multivocal literature review (MLR), supplemented by interviews with software practitioners and academics. The results are complemented by findings from our preliminary study (Tom et al., 2012) which includes a theoretical framework that addresses the nature of technical debt itself as well as its precedents and outcomes. The outline of this article is as follows. A brief background to the study is presented in Section 2. The MLR process and interviews are detailed in Section 3. The results of the MLR and interviews are presented in Section 4 followed by discussion in Section 5. Section 6 presents conclusions drawn in this research.

2. Background

Technical debt is a metaphor that refers to the consequences of poor software development. Cunningham (1992), who introduced the concept of technical debt, described how "*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*". Since then, the suitability of debt as a way of explaining the various drivers of increasing costs throughout the life of a software system has been affirmed by the software development community (Guo and Seaman, 2011; Klinger et al., 2011; Seaman et al., 2012; Lim et al., 2012; Snipes et al., 2012). On the other hand, debt is not necessarily 'bad' – a small level of debt can help developers speed up the development process in the short term (Guo and Seaman, 2011). In any case, the consequence of this may be felt in the longer term if the project is highly 'geared' (which implies onerous debt repayments), leading to slower development and killing of productivity.

Based on extant academic literature we developed a systematic literature review (SLR) (Kitchenham, 2004) in June 2011 on technical debt in order to understand the state of the research addressing this area (Tom et al., 2012; Tom, 2011). The results reveal that there is a limited numbers of articles that refer to the concept of technical debt.

The findings (Tom et al., 2012) revealed an apparently fragmented understanding from a number of disparate themes and anecdotal or otherwise high-level definitions that fail to describe the phenomenon comprehensively or in detail. For example, Brown et al. (2010a) propose that technical debt can be characterised as the "*gap between the current state of a software system and some hypothesized 'ideal' state in which the system is optimally successful in a particular environment*", suggesting that known defects, unimplemented features and outdated documentation can all be considered aspects of debt. Whilst visualising technical debt as a gap is a useful conceptual tool, it is essentially another metaphor, and does not further clarify what actually constitutes technical debt in a literal sense. However, the additional purported dimensions of debt provide insights into what the collective understanding of technical debt (by the academic and practitioner communities) might include. Other examples of such definitions include "*the degree of incompleteness*" (Klein, 2005), "*a backlog of deferred technical problems*" (Torkar et al., 2011), and "*any side of the current system that is considered sub-optimal from a technical perspective*" (Ktata and Lévesque, 2010). Whilst more specific definitions such as "*bugs, design issues, and other code-quality problems*" (Black et al., 2009) can be found in the literature, a comparison of the specific dimensions of technical debt proposed in each definition reveals disparity. This is affirmed by Brown et al. (2010a) who recognise that, beyond what is intuitively understood, "*a more rigorous definition and validation of the* [technical debt] *concept, and the heuristic practices it implies has not been undertaken*". A list of explicit and implicit definitions of technical debt from the SLR is illustrated in Appendix C.

The findings showed that whilst code decay and architectural deterioration are commonly recognised to be major dimensions of technical debt, accurately communicating the extent of technical debt in a system poses a significant challenge. It is evident that the boundaries of technical debt, as reflected in academic literature, are *fuzzy* – they lack clarity and definition – and represent a barrier to efforts to model, quantify and manage technical debt (Brown et al., 2010a).

The SLR findings showed that technical debt can be holistically represented by the "precedents" and "behaviours" that cause the phenomenon, "metaphorical" and "literal elements" that constitute the phenomenon's intrinsic nature, and a number of "outcomes" that result from healthy and unhealthy forms of technical debt (Tom et al., 2012; Tom, 2011). The findings also showed that numerous constructs form the representation of technical debt in academic literature and some of these constructs are better supported in literature than others. Appendix B shows constructs from this SLR. The list of academic references from the SLR are illustrated as "*A#" in the Reference section.

Whilst code decay and architectural deterioration are commonly recognised to be major dimensions of technical debt (Eick et al., 2001; Shull, 2011), many other concepts including a lack of documentation and testing are yet to be widely recognised and agreed upon as forms of debt (Shull, 2011; O'Connor, 2010). It became apparent from the literature that recognising, quantifying and accurately communicating the extent of technical debt in a system poses a significant challenge. Time as a constraint is a recurring theme in the technical debt literature (Thibodeau, 2011), being impacted by, and at the same time a factor of, technical debt. Other project constraints such as budgeting and resourcing are also drivers for allowing technical debt to accrue, assisted by the low visibility of the phenomenon (Ktata and Lévesque, 2010). The notion of debt as a type of investment that enables an increase in development velocity in the short term creates the temptation to allow it to accrue, despite it inevitably slowing velocity and increasing time-to-market in the long term if it is not repaid. Thus, it becomes important to gain a complete understanding of technical debt and the actual trade-offs (obvious and otherwise) that are made when it is allowed to accrue. This importance is emphasised by risks comparable to interest and potential bankruptcy, along with the literal consequences (good and bad) underpinning the debt metaphor.

3. Research methodology

Findings from the preliminary study show that the academic literature is lacking a comprehensive definition or conceptual model of this phenomenon (Tom et al., 2012). However, other accessible writings (e.g. internet blogs, white papers, and trade journal articles) termed "*multivocal literature*" (Ogawa and Malen, 1991) readily propose definitions, measures, motivations and ideas surrounding the concept of technical debt. These writings provide a method for exploring the concept of technical debt. On the other hand, there are apparent issues of reliability and validity associated with these writings due to their diversity. Whilst this means that the information presented in multivocal literature cannot be immediately assessed in the same way that academic writing would be, both the contemporary and interpreted nature of the technical debt phenomenon would suggest that the value of multivocal literature should not be overlooked.

This research uses our previous preliminary study of the academic literature (Tom et al., 2012) as a guideline. In this article, we incorporate data collected from practitioners – both primary

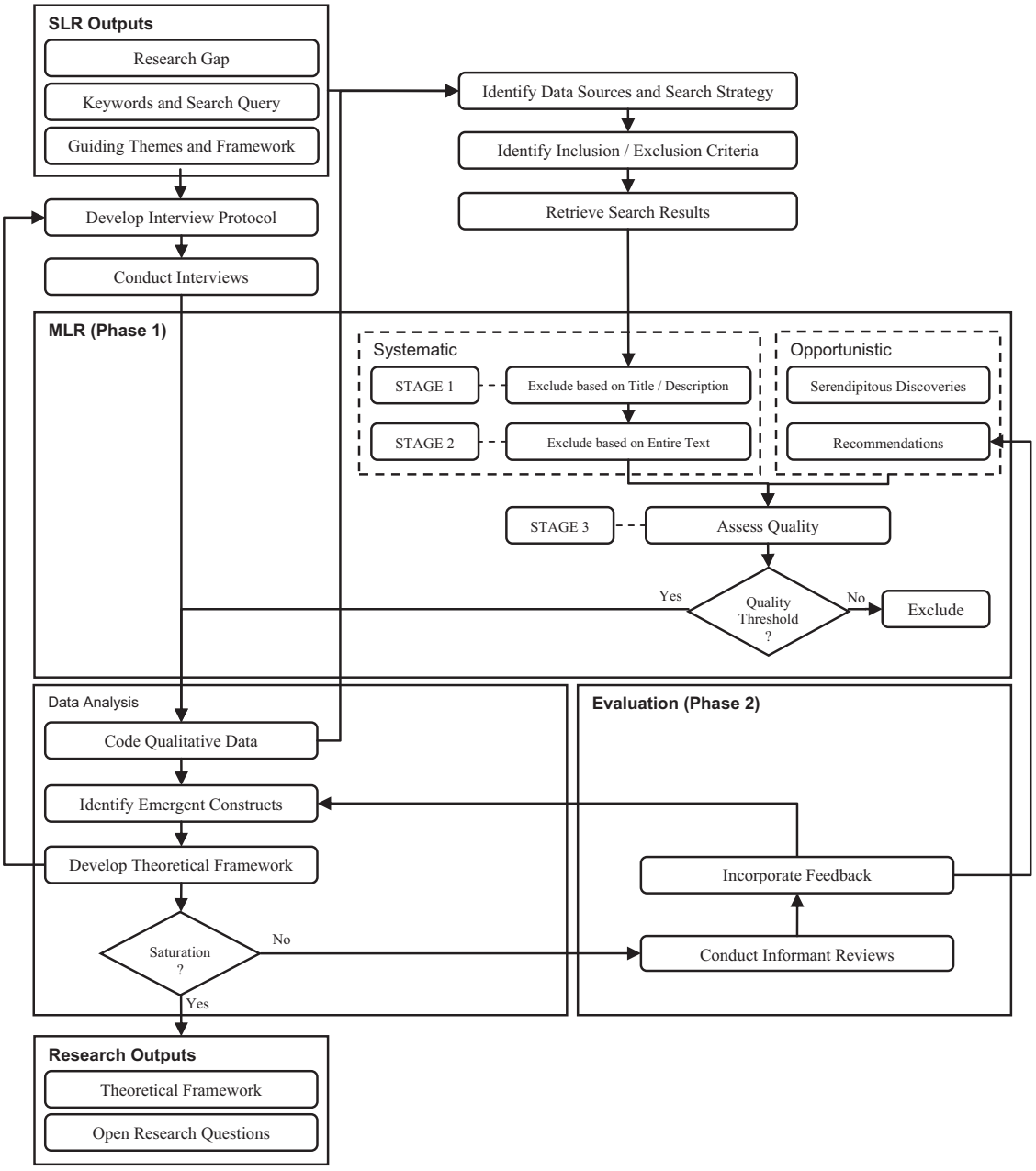

Fig. 1. Research method.

and secondary, in the form of online, non-academic literature through MLR – and then evaluate the resultant framework through interviews with academics and software practitioners. Thus, this study is carried out in two phases. In phase-1, MLR is conducted followed by semi structured interviews with software practitioners/academics. From the analysis of data collected from MLR and interviews a theoretical framework is developed. In phase-2, informant reviews are conducted with practitioners to evaluate the framework.

MLR applies a qualitative review approach to engender rigour when systematically reviewing multivocal literature – a term that collectively refers to other accessible, non-academic writings on a topic (Ogawa and Malen, 1991). MLR, as SLR, also uses a qualitative research approach in systematic reviews, by developing a search strategy and applying appraisal criteria, and then synthesising data from multiple stakeholders. Ogawa and Malen (1991) argue that MLR can be considered to be a "*case study of representations of the topic*", whereby each document in the study is treated as a separate data point. This method allows documents to be as diverse as the data points of a case study, yet still able to be aggregated and analysed to develop conclusions about the literature and inferences about the phenomenon. In contrast to any alternative research methods, the Ogawa and Malen's (1991) approach provides the opportunity to conduct a holistic inquiry leveraging the full extent of accessible writings.

Fig. 1 summarises the steps of this research protocol, including the MLR and interview components. As seen in Fig. 1, the research protocol incorporates several feedback loops to enable an iterative approach to a number of processes, including developing the interview protocol, finding search terms for the multivocal literature review, discovering texts and developing a theoretical framework from qualitative coding.

3.1. Phase 1: MLR and interviews

3.1.1. MLR

The MLR is conducted in three stages as shown in Fig. 1.

1. *Data sources and search strategy*: Google's web search engine (http://www.google.com) was used to source multivocal literature from the World Wide Web. The SLR provided a preliminary set of keywords referring to the phenomenon of interest and a suitable search query. This query was used as the initial input into the search engine to retrieve results matching any one of the following terms – "technical debt", "design debt", or "debt metaphor". Any other terms found to describe the technical debt phenomenon during the qualitative coding stage of the review were used to update the search strategy, with results being retrieved in an additional iteration, and new documents being assessed for inclusion in the multivocal literature review. The first 50 relevant articles (see Appendix A), as determined by Google's ranking algorithm, is taken as the data source for subsequent analysis.
2. *Inclusion and exclusion criteria*: The records were reviewed against a set of inclusion and exclusion criteria. Search results were excluded if they were (a) a duplicate record; (b) did not relate to the process of software development; (c) did not discuss the concept of technical debt.
3. *Quality assessment*: The results were assessed for quality in terms of the position and certainty of the source, clarity, detail, consistency, plausibility, and alignment with the research focus as determined by the research questions. Results deemed not to meet a minimum quality threshold were excluded at this stage of the selection process. Such a threshold could not be predetermined due to the diverse nature of MLR, and case-by-case assessment and consideration were necessary. In practice, quality assessment was an iterative process whereby results were divided into three categories during each iteration – above the quality threshold, uncertain, below the quality threshold – until no results remained in the uncertain category.

3.1.2. Interviews

Semi-structured interviews were conducted in parallel with the MLR. A total of 11 interviews were conducted with participants identified to reflect a range of roles associated with the software development industry who were experienced in software development and currently dealing with the problems associated with technical debt. Interview participants were recruited from a convenience sample of academics and practitioners who were contacts of the authors of the study.

The interview questions covered participants' background, dimensions and causes of technical debt, and its benefits and drawbacks. All interviews except one were conducted face-to-face – one interview was conducted via a Skype call due to geographical separation between the interviewer and interviewee. Interviews had an average duration of approximately 30–45 min. The interview schedule is provided in Appendix D.

Interview transcripts and results from the MLR were coded using the computer-based qualitative tool NVivo following Miles and Huberman's (1994) guidelines. Codes were assigned inductively, and a constant comparative method was employed in an iterative approach that allowed already identified codes and themes to be tested against the remaining data set (Silverman, 2000). Each iteration of coding was archived as a copy of the NVivo project file to preserve the history of code renaming, reassignment, splitting to identify finer-grained themes, and lumping to identify over-arching themes. Following the coding of raw data, codes were consolidated into hierarchical categories for the purpose of identifying emergent constructs, helping to increase the manageability and discernibility of seemingly disparate sources (Ogawa and Malen, 1991). Findings were then organised into a theoretical framework for discussion at informant reviews.

3.2. Phase 2: informant reviews

The objective of informant reviews was to confirm and challenge the framework that was developed from the analysis of results from the MLR and interviews. Three informant reviewers participated at this stage. Their roles were build engineer, contract researcher, and senior software engineer. The informants were given a document (~7500 words), via email, that included the results of our three research question from the MLR and 11 interviews, including the dimensions of technical debt, attributes of technical debt, a draft technical debt taxonomy and a draft theoretical model. These informants were asked to review this document and the preliminary version of the theoretical framework described by the research results with regards to the following points:

(a) Aspects the informant disagreed with;
(b) Aspects the informant believed required more emphasis or discussion;
(c) Aspects the informant believed missing;
(d) Other comments and general feedback.

The feedback from the informants was gathered as a text comment on this document. This feedback was then incorporated into the theoretical framework via an iterative process that involved revisiting the identification and assessment of emergent constructs, and further developing the theoretical framework as appropriate.

4. Results

Table 1 summarises the results of the selection process for the MLR.

Table 2 provides a summary of interview participants across roles and years of experience related to the software development industry.

In the analysis of the three research questions, the following sections examine the results of MLR, eleven interviews, and feedback from the three informant reviews.

Table 1
Multivocal literature review selection process.

	Initial	Stage 1	Stage 2	Stage 3
First iteration	50	50	45	32
Second iteration	50	15	14	3
Total				35

Table 2
Interview participants across roles and years of experience.

Role	Years of experience				Total
	0–5	6–10	11–15	16+	
Software Developer/Engineer	1	1	2	–	4
Build/Performance Engineer	–	–	1	1	2
Product Architect	–	–	1	–	1
Development/Construction Manager	1	1	–	–	2
Researcher/Academic	–	1	–	1	2
Total	2	3	4	2	11

4.1. Dimensions of technical debt

The following provides the results of the analysis of the dimensions of technical debt (RQ1). The results are synthesised by combining the MLR, the interview participants' perspective, and the feedback from the three informant reviews. The source of the evidence is referenced throughout the section.

4.1.1. Code debt

Technical debt is often manifest in the form of poorly written code – "*every hack, work around, bad piece of code builds technical debt*" (Stopford, 2010). Examples provided by interview participants include unnecessary code duplication and complexity, bad style that reduces the readability of code, and poorly organised logic that makes it easy for a software solution to break when updated at a future point in time. In essence, code that requires refactoring (and where this refactoring does not fall into the category of design) is a form of code debt.

4.1.2. Design and architectural debt

Design and architecture shortcuts and shortcomings were found to be another constituent of technical debt. Design debt can be in the form of upfront design with an under-focus on qualities such as maintainability and adaptability, or subsequent piecemeal design with an absence of refactoring (Shriver, 2010). Similarly, architectural debt could be the result of sub-optimal upfront solutions, or solutions that become sub-optimal as technologies and patterns become superseded.

4.1.3. Environmental debt

Technical debt can also manifest in the environment of an application, which includes development-related processes as well as hardware, infrastructure and supporting applications. Nielsen (2011) says "*Get a sense of the daily, weekly, and monthly operational tasks... how many of your teams are accountable for manual operational tasks? What are those tasks? What is the frequency?*" Manual processes with the potential to be automated are a form of technical debt that accrues interest in terms of manual labour costs, and the opportunity cost of any superior capabilities offered by automation. Outdated components of an application's development or operating environment also contribute to technical debt. Examples include the security vulnerabilities that may be exposed by the postponement of upgrades to infrastructure (which can accrue high technical debt "interest charges" in the form of brand damage), and components that have moved to an end-of-life status (which accrue interest in the form of maintenance costs).

4.1.4. Knowledge distribution and documentation debt

A lack of knowledge distribution is another aspect of technical debt. Slinker (2008) gives the following example in a blog post: "*A large system with millions of lines of code may be maintained inexpensively. One factor that keeps the expense down is that the original developers stay on with the system. They know why and how things were done. Thus code debt is affected by the members of the team. Suppose the team becomes insulted in some manner and all quit. Suddenly the [technical debt] changed from low to extremely high!*". Interview participants also gave examples of technical debt consisting of the absence of well-written documentation.

4.1.5. Testing debt

Technical debt can take the form of a lack of test scripts leading to the need to manually retest the system before every release, or insufficient test coverage regardless of whether tests are automated or manually run. Testing debt can incur increased costs associated with brand damage when customers are affected by critical defects, the need to implement fixes in a production environment, the need to diagnose and fix regressions, and the need to execute manual testing when automated test scripts are not available.

4.2. Attributes of technical debt

The analysis of MLR and interview transcripts revealed additional facets of technical debt's intrinsic nature – apart from a set of literal software development dimensions in which technical debt is manifest. The phenomenon can also be considered in terms of specific attributes as explained below. Most of these codes described attributes that reflect analogies which can be drawn between the technical debt phenomenon and facets of financial debt.

4.2.1. Monetary cost

This study finds that technical debt ultimately has negative impacts on morale, productivity, quality, and risk. As a result of these outcomes, technical debt is not only a financial *metaphor*, but is in fact associated with a real monetary cost. Developer time is expensive and is ineffectively utilised when development velocity is slowed and developers are forced to fix regressions rather than develop new features. Additionally, a direct monetary cost can be seen when quality issues result in "*customers returning the product and the lack or loss of sales*" (Slinker, 2007). Myers (2009) presents similar arguments on his blog to support his view that "*in effect, design debt is real debt*".

An example of a less direct, yet equally significant financial impact of technical debt is given on Laribee's (2009) blog – "No developer enjoys sitting down to his computer in the morning knowing he's about to face impossibly brittle, complicated source code. The frustration and helplessness thus engendered is often a root cause of more systemic problems, such as developer turnover – just one of the real economic costs of technical debt". In line with this view, interview participants expressed the sentiment that there is an increasing focus on explicitly associating costs with technical debt – that is, whenever a decision (direct or otherwise) is made to accrue technical debt, there's an increased focus on recording that debt and the costs that can be attributed to those decisions.

Despite this, there is a challenge in quantifying technical debt in any of its forms. Interview participants noted that although some forms of debt have more obvious monetary aspects, no metric exists to assist with quantifying technical debt and calculating monetary cost. One performance engineer likened it to asking "*how much faster* could *you have been?*", when there's no way of meaningfully proving what developers intuitively know to be a significant amount of technical debt, particularly with the many confounding variables in place – "*maybe I could have done something faster, but would it have had more bugs or less bugs, and what are the bugs and do they matter?*".

4.2.2. Amnesty

Ries (2009) encourages readers of his blog post to "*invest in technical debts that may never come due*" as one way of embracing the technical debt concept. Debt amnesty – where accrued technical debt can be thought of as written off and does *not* have to be repaid could come about in situations where a throwaway prototype is being developed, or when a feature or product is deemed a failure and no longer needed (for reasons other than technical debt – such as a lack of customer interest or value) (Ries, 2009). Technical debt is also "*retired*" along with a system at the end of the system's service life (McConnell, 2007), effectively resulting in debt amnesty. Although interview participants concurred that debt amnesty is one possible scenario following the accrual of technical debt, they were quick to warn that this should not be used as an excuse to recklessly accrue technical debt – "*you might want to re-use the code, and you end up rewriting the same pieces of code over different projects for different clients. Or at the same time, a client might go 'that was*

really successful, could you do it again?', except this time they might also want this and that, and you need to go 'well, that's going to take just as long as the original thing because of how quickly we did it'".

4.2.3. Bankruptcy

The concept of bankruptcy as it is applied to technical debt refers to a situation of overwhelming debt interest, whereby progress is halted and a complete rewrite is deemed necessary. As technical debt is difficult to quantify and situations vary, there is no precise point at which technical debt bankruptcy should be declared. In his blog, Elm (2009) describes an application to be bankrupt when "*the cost of improving the existing code in an application may indeed be greater than the cost of rewriting it*", whilst Hilton (2011) implies that bankruptcy is the point at which "*you will either suspend all feature development to pay down all of your debt at once, or you will have to rewrite the entire application*". Regardless of how technical debt bankruptcy is defined, there are challenges around its interpretation. A number of interview participants warned that from their experiences, the point at which bankruptcy is declared is often much earlier than a true point at which a rewrite becomes necessary. A software engineer noted that "*it's quite common for people to want to do a rewrite because 'the old code's crappy, I don't want to deal with it anymore'*". Interview participants were able to give several examples of system rewrites they had experienced, and agreed that bankruptcy is a possibility when tactical, incremental and inadvertent forms of technical debt (which do not have a view to the long term) are left unmanaged without any controls in place. In some cases, bankruptcy was reached when "*adding new features took longer than it would take to rewrite*", whilst in other examples performance and scalability improvements were no longer practical without a rewrite – "*the solution really started off as being a proof-of-concept, which then became a production environment... it wasn't scalable and the data model wasn't designed for volume*".

4.2.4. Interest and principal

A fundamental characteristic of financial and technical debt in all its forms is the notion of interest payments – "*every minute spent on not-quite-right code counts as interest on that debt*" (Cunningham, 1992), which needs to be paid in addition to principal repayments. Camden (2011) explains that in software development, paying back the principal is a matter of completing a deferred task (or implementing a correct replacement for a shortcut), and interest payments are the costs associated with the workaround or deferral. More broadly, interest payments on technical debt are associated with its impacts on morale, productivity, and quality. For example, low developer morale could lead to a high rate of turnover (Laribee, 2009), which in turn exacerbates the problem by increasing technical debt – particularly with regards to the distribution of knowledge. Decreased productivity in itself is an interest cost, but it also follows that slower development velocity provides an increasing incentive to prioritise – potentially causing more debt. Managing quality issues such as defects (regressions in particular), maintainability and extensibility problems affects productivity, and these issues cost the business directly when they influence sales and brand image. A noteworthy aspect of the interest attribute is that different interest rates also exist in the context of technical debt. In his blog post, Eric Ries (2009) explains that "*if a given feature is rarely modified, its debt is much less expensive*". Technical debt that exists in components that are rarely revisited incurs lower interest than what Ries (2009) describes as "*debt in a core system*", where technical debt is much more likely to "*manifest as intermittent defects all throughout the product, each of which is hard to localize and debug*".

Ries's (2009) example was further elaborated during the informant review stage of this research, with a software engineer remarking that "*this is only half the story, as it also depends on how frequently users actually use the indebted feature*". The software engineer explained that a debt-laden feature might be rarely modified by developers, but could frequently impact users – in this case "*users have to find and use a workaround, which will cost real money in support (for motivated evaluators/customers), sales (through turned-off evaluators), and renewals (through dissatisfied customers)*".

4.2.5. Leverage

Technical debt is often deliberately created for leverage and can increase productivity in the short term. This allows teams to strategically or tactically take shortcuts and defer work to accrue technical debt in exchange for "borrowed time". Fowler (2009a,b) describes a technical debt payoff line, below which technical debt can be effectively leveraged to deliver a greater amount of functionality relative to time. There is a trade-off between quality and short time-to-market, and accruing technical debt to deliver early can yield greater payoffs early on. However, Fowler (2009a,b) warns that "*you need to deliver before you reach the design payoff line to give you any chance of making a gain on your debt... even below the line you have to trade-off the value you get from early delivery against the interest payments and principal pay-down that you'll incur*". This notion of leverage can also be considered in terms of managing a debt ratio, whereby if the ratio of debt to the system as a whole is too high, the company will need to invest all its efforts into servicing that debt (e.g. keeping a production system running) rather than increasing the value of its other assets (e.g. by adding new capabilities to the system) (McConnell, 2007). This sentiment is shared by interview participants – a product architect remarked that "*a certain amount of debt is not only inevitable but probably a good thing... as long as you have it as a manageable quantity and know it's there*".

4.2.6. Repayment and withdrawal

Technical debt can be characterised in terms of its repayment and accrual. Aspects of technical debt's repayments and withdrawals are characterised by "*hundreds or thousands*" of small withdrawals comparable to purchases made with a credit card (McConnell, 2007). Whilst this type of debt involves a conscious decision (at least on the part of an individual developer), it is much easier to accumulate unintentionally, and is "*much harder to track and manage after it has been incurred*" (McConnell, 2007). McConnell also suggests that the concept of credit ratings can be associated with the withdrawal of technical debt – by assisting teams in deciding how much deliberate debt is appropriate to take on. McConnell (2007) describes a team's credit rating as a reflection of "*a team's ability to pay off technical debt after it has been incurred*", which might consider factors such as the level of debt that a team has already taken on, and whether the team's velocity has started to drop as a result of technical debt. Different forms of technical debt are also characterised by the nature of the repayments that need to be made. For example, in some cases technical debt is much less visible (due to the size of the debt being considered too trivial to identify and track), making it more difficult to recognise the need to make repayments in the first place. Interview participants suggested that these forms of debt are less frequently repaid, and that yet another reason might be that repayments are often particularly "*not fun*", as they more often involve "*mundane*" refactoring rather than the implementation of new functionality.

4.3. Why organisations take on TD

RQ2 investigates how technical debt arises. During data analysis, a number of codes describing precedents of technical debt were identified. These codes represent the factors that contribute to the accrual of technical debt, which are categorised into a number of top-level precedents.

4.3.1. Pragmatism

Pragmatism and prioritisation are strongly linked – one invariably, though perhaps unconsciously, prioritises when being pragmatic, whilst effective prioritisation also involves a level of pragmatism. However, in this study, pragmatism and prioritisation are considered two separate precedents of technical debt, to distinguish between the directly external drivers of prioritisation (e.g. project constraints), and the internal, or indirectly external influences of pragmatism. Haack (2005) notes in a blog post, "*sometimes you just have to bite the bullet and weigh business needs against design purity and just spew code*". In this case, the code is poorly written, with small and numerous instances where refactoring is necessary. It can be easily seen that pragmatism can also lead to the deferral or sub-optimal implementation of larger, more identifiable tasks.

An example of pragmatism, creating a minimum viable product in a short amount of time, is noted by Brazier (2007) who says in his blog post that the futures of small companies in niche markets often rely on being first to market – "*longer term software problems simply aren't important in this case because they are not visible to the customers until it's too late... and they are certainly less important to the software company for whom the long term won't exist unless it makes the sale in the short term*". This example is supported by interview participants with experience in a bootstrapped software company. As one product architect recounts, "*it was vitally important that the product got into customer' hands as early as it could, and that those customers gave us money for that product, so that we could then use that money to pay developers to keep working on it*".

4.3.2. Prioritisation

Prioritisation is another precedent of technical debt. When it becomes necessary for developers and teams to prioritise their tasks, the implementation of critical functions are generally prioritised above overall quality. In order to deliver required amounts of functionality within specified project constraints, teams will need to make tradeoffs in aspects other than functional requirements. These tradeoffs often create technical debt.

The project constraints most often discussed by interview participants and multivocal literature as precedents to technical debt are time, budget, and resource constraints. With regards to budgeting in particular, there is often a push to reduce capital costs as much as possible. McConnell (2007) suggests that in a startup environment, preservation of capital is important because "*every dollar counts... if you can delay an expense for a year or two you can pay for that expense out of a greater amount of money later than out of precious startup funds now*". Another motivation described by a performance engineer is that "*people want to keep the capital cost of the initial software development down, because the maintenance costs come out of a different budget*".

Even without direct project-related constraints, expectations from customers and management can influence the accrual of technical debt. Hilton (2011) gives an example in his blog post – "*Maybe for the first 12 months of development on the product, your team worked at a certain pace. Your customers, your product owners, and the rest of your organization came to understand this pace, and have now come to expect it. Unfortunately, since the 8th month or so of the development, the only way to keep this pace has been to cut corners*".

Interview participants noted that as requirements change, expectations are increased. A construction manager explained that in some cases there are internal business stakeholders "*trying to drive [a project] in different directions, or trying to add complexity or remove complexity in some cases*". When these requirements and expectations from customers and management are combined with project constraints, there is a strong need for developers to prioritise aggressively.

A more extreme case of customer or management direction as a precedent for technical debt is the scenario where a decision is made to adopt a prototype as the working product. Interview participants were able to recount such cases when speaking of their experiences. A software engineer said "*I've seen some projects where we spend a few months putting together what was essentially meant to be a throwaway prototype of the system... management then decided that it was to be the product and we would then have to maintain that*", and a construction manager described an example where "*the solution really started off as being a proof of concept, which then became a production environment... it's hard to explain to the business stakeholders that a proof of concept is just a proof of concept*".

4.3.3. Processes

The processes adopted by a team can also affect the amount of technical debt that is accrued. The visibility of all forms of technical debt decreases when poor communication and collaboration processes are in place, making it easier for debt to accumulate without being noticed. Shriver (2010) suggests the use of daily stand-up meetings, task boards, and burndown charts to increase communication and reduce technical debt in low-cost ways. When developers are not aware of the amount of overall technical debt in a system, it's much easier to make individual decisions to take shortcuts and defer tasks that increase technical debt. A lack of the processes that will usually discourage developers from creating technical debt will also contributes to its accrual. Interview participants identified code reviews as such a process. As explained by one of the software engineers, "*if you're writing code knowing that someone else is going to be reading it very shortly, and they're going to have a lot of questions, you're much more likely to at least make it self-explanatory*". These processes can also ensure that technical debt is identified and addressed early. With regards to code reviews, the software engineer explained that "*after a bit of conversation, you may be more likely to make it simpler and more likely to try to eliminate some accidental complexity that's crept in*".

4.3.4. Attitudes

This study finds that individual attitudes play a significant role in contributing to technical debt. Elm (2009) describes one of the common reasons for unmanaged debt to be "*programmer and management hesitance to improving code, for fear of introducing new problems: 'If it ain't broke, don't fix it'*". Interview participants noted attitudes that could best be described as general apathy towards issues of technical debt and software quality as a factor in technical debt accrual. These attitudes influence individual decisions to create a technical debt, and can also bring about a higher level of carelessness. One example given by a software engineer is that "*if you have a lot of Java developers who are very experienced in java and don't care about the JavaScript they're writing, most of the JavaScript is going to wind up being a form of technical debt*". Additionally, risk appetite at an individual, team or organisational level can influence decisions related to technical debt in different ways. Technical debt can influence risk positively or negatively in the short term, so having a lower risk appetite can influence decisions that create technical debt to reduce a project's delivery risk in the short term, and having a higher risk appetite can influence decisions to create technical debt regardless of the fact that it may be increasing project risk.

4.3.5. Ignorance and oversight

Whilst oversight is a form of ignorance, in that developers are unaware of the mistake or issue that is creating technical debt, ignorance refers to ignorance of *how* to avoid technical debt rather than ignorance of its presence. Ignorance might refer to "*the inability of the developers to develop high quality applications*" (Cast, 2011), where individual developers or entire teams do not have adequate knowledge on "*how to write clean code, their business domain and how to do their jobs optimally*" (Shriver, 2010). Ignorance might also

be much more specific to a particular task – for example the optimal design pattern to use in a specific situation. Interview participants have suggested that technical debt due to oversight can arise when developers are "*not being methodical about the code [they're] writing*", "*trying to do something without considering the ramifications of the change they're trying to make*", and "*not thinking about what else is going to be built on top of the changes that are made*".

4.4. The outcomes of TD

RQ3 investigates the consequences of technical debt. According to the data collected from multivocal sources and interviews, the accrual of technical debt impacts four top-level outcomes – team morale, productivity, quality of the product, and project risk. This study finds that allowing technical debt to accrue can result in some short term benefits – whilst quality will likely decrease, in the short term morale *may* increase, project delivery risk *may* decrease, and productivity *may* increase. However, technical debt that has not been repaid in the long term impacts *all* outcomes negatively, in the sense that morale, productivity and quality are decreased whilst project risk increases.

4.4.1. Impact on morale

Addressing technical debt is a mundane task for many developers, and it seems likely that developers would find the effects of technical debt frustrating in the long term. Laribee (2009) explains that – "*beyond the obvious economic downside, there's a real psychological cost to technical debt... this psychological effect takes a toll on team morale*". The impact of technical debt in the short term varies depending on individual preferences and the situation. Interview participants explained that a lot of the tasks to prevent or repay the accrual of technical debt "*aren't very fun*", so allowing technical debt to accrue so that developers can continue working on adding features and taking on more challenging tasks can in fact increase short term morale for some individuals. However, for developers who take the quality of the code they write very seriously, technical debt has a negative impact on morale, even in the short term. In the long term, developers are forced to repay interest *and* principal on accrued debt so that they can continue adapting and maintaining their system. As a result, technical debt has a strongly negative impact on morale in the long term – "*morale deteriorates as technical debt continues to mount with each short cut taken without a view toward ending the vicious cycle*" (Shriver, 2010). Atwood (2009) considers this incremental form of technical debt to be particularly frustrating in the longer term.

4.4.2. Impact on productivity

The main benefit of technical debt is that it can increase development velocity in the short term by allowing developers to leverage the time-savings afforded by shortcuts, deferrals, and other trade-offs. One of the software developers explains that "*often, developers' time may be far more valuably spent on something that's going to generate immediate customer value than on something later on, and obviously the younger a product or company, the more value there is in doing something immediately*".

The reason that technical debt is considered to "*kill productivity, making maintenance annoying, difficult, or, in some cases, impossible*" (Laribee, 2009) is because it is analogous with a high-interest loan that makes future changes more expensive (Fields, 2011). When not repaid in a timely manner, technical debt requires interest payments which decrease long term development velocity in a number of ways.

First, the impacts of existing technical debt on structural quality make the existing codebase harder to modify. In his blog post on why technical debt matters, Shalloway (2011) elaborates – "*This means that even when we work on new features, it'll take longer to get them done. So, as code quality gets worse, we not only have less time to work on new features, but it'll take longer to get them done because the code will be harder to change. Of course, what often happens is that the developers just hack things in to meet deadlines. This, of course, just gets the code quality to go down faster. Bad cycle here*".

This sentiment is shared by the interview participants of this study, who explain that a slowing of the speed of development is one of the first noticeable impacts of technical debt. One of the reasons for this decrease in velocity is the increased likelihood of regression defects in systems with a high level of technical debt – "*code that is not properly maintained is inherently more brittle*" (Elm, 2009). When developers are spending additional time diagnosing and fixing regressions, they have less time to implement features.

Second, velocity is also decreased in the long term simply because developers are spending time managing and working around technical debt issues. For example, when there is inadequate knowledge distribution and documentation, developers need to spend more time familiarising themselves with the codebase and specific problem domain. A software engineer noted that "*implementing workarounds for issues in the code can take significant time, discovering that you have to implement a workaround in the code can take longer, and it can be quite difficult to find reference information that can tell you that if your thing has* this *problem, then you need to do* this *thing*".

4.4.3. Impact on quality

Technical debt has a negative impact on product quality in terms of defects and other structural quality issues. Technical debt in all its forms creates quality issues in the short term, but as technical debt accrues into the long term, quality is further impacted as a result of interest costs. Technical debt contributes to both the creation of defects and the fact that defects remain unnoticed in a product. When code is poorly written such that it is difficult to follow and easy to misunderstand, or when sub-optimally designed solutions didn't consider all cases, it is a lot easier for defects to arise.

Discovering defects is also more difficult when the technical debt in a system relates to a lack of tests or test coverage. This is a significant outcome – Myers (2009) claims that the number and severity of defects experienced by users has a "*fairly direct relationship with the financial impact of lost sales, time spent on support calls, developer time spent debugging the product, and on and on (and on)*".

This study also finds that the quality attributes most frequently compromised when trade-offs that introduce technical debt must be made are the structural attributes of extensibility, scalability, maintainability, adaptability, performance and usability. It should be noted that these quality attributes were identified as codes following an open coding approach during the data analysis phase. During the informant review phase of the research, a software engineer remarked that other quality attributes likely to be impacted by technical debt include testability, supportability, reliability and security.

4.4.4. Impact on risk

The impact of technical debt on project risk is closely related to its effect on productivity. Shriver (2010) explains that when there's a high level of technical debt creating uncertainty around making any changes, "*it's extremely hard for even the best developers to estimate... increasing the likelihood of late deliveries*". The impact on project risk is further exacerbated when existing technical debt is not visible. This is often the case with inadvertent debt when it has not yet been recognised as debt, and incremental debt that is difficult to track. A software engineer likened these forms of debt to "*unknown unknowns*", and "*known unknowns*" respectively, whilst strategic and tactical debt that a team is aware of can be considered a third, "*known knowns*", category. This interview participant explained that "*each of those [categories] is a risk to getting a*

product shipping on time... just being able to discover that [inadvertent] technical debt and reduce the unknown unknowns (and move it into the known unknowns) helps". However, and perhaps counter-intuitively, allowing technical debt to accrue also has the potential to *decrease* project risk in the short term. A development manager used the phrase "*I already have a lot of moving parts, I don't want to add another*" to illustrate an example where allowing technical debt to accrue in a release that was already high-risk prevented further risk with regards to meeting timelines. The manager made a decision *not* to implement a new technology across all business functions, which was a form of technical debt because some of these functions now had an alternative (lower-risk) implementation, which would have to be revisited and re-implemented at a future point in time to restore consistency in the system's overall design.

5. Discussion and implications

This section discusses the findings and the implications for practice and research. As data analysis progressed distinct types of technical debt with different characteristics were revealed in the data. We decided to adopt McConnell's (2007) classification when explaining the different groups of technical debt and renamed McConnell's classification for clarity as strategic, tactical, incremental and inadvertent technical debt.

5.1. Strategic debt

Strategic debt refers to what McConnell (2007) describes as "*long-term debt, usually incurred proactively, for strategic reasons*". One example of strategic technical debt is the proliferation of trade-offs that ignore the potential for long-term software problems in favour of increased development velocity in the short term (for example, with the aim of being first to market). This scenario is strategic when it is either crucial for the company's continued existence, or when there is only a small window of opportunity for new implementations to arrive on the market (Brazier, 2007). Key aspects of strategic debt are that it is accrued for the purposes of leveraging borrowed time, the debt is sizable, more distinct, more visible, and the debt repayment is in the long term.

5.2. Tactical debt

Tactical debt is also sizable, distinct, and visible – the key difference between strategic and tactical debt is that the latter is taken on in a reactive manner for the short-term, "*usually as a late-stage measure to get a specific release out the door*" (McConnell, 2007). In this manner, it also leverages borrowed time. Examples of tactical debt include sub-optimal solutions – "*we don't have time to implement this the right way; just hack it in and we'll fix it after we ship*" (McConnell, 2007) – and deferred work that can be distinctly identified as tasks, such as a library or framework that needs to be upgraded.

5.3. Incremental debt

McConnell (2007) describes the incremental form of technical debt as "hundreds or thousands of small shortcuts – generic variable names, sparse comments, creating one class in a case where you should create two, not following coding conventions, and so on". McConnell (2007) likens this form of technical debt to credit card debt that easily accrues, and is difficult to track and manage once incurred because withdrawals are small and numerous. Interview participants discussed a lack of refactoring as a key contributor of technical debt, explaining that incremental debt is an important factor in increasingly costly maintenance as a system gets older. A development manager remarked that "unless a manager of [a maintenance team] consciously makes a decision to continue to refactor the code in the right way – if they just keep adding more and more functionality and don't come back to revisit key design decisions made more than 6 years ago, eventually that's going to cost you more and more money to maintain".

Inadvertent Debt: Inadvertent debt is simply the form of technical debt that accrues unintentionally – ultimately due to ignorance and oversight. Interview participants recognised that whilst this form of debt is not entered into consciously, it still accrues interest and needs to be repaid. One example given by a development manager in the interviews is that "*a developer may have made a decision not knowing the impacts of their decision because they've done the change in isolation*" – when these impacts have the potential to create increasing costs for the future, inadvertent technical debt has arisen.

Table 3 summarises proposed associations between technical debt's software development dimensions, attributes, precedents, outcomes, and the different forms of debt in the proposed taxonomy. Potential associations are indicated by a solid circle. These associations are conjectured using examples from MLR and interview data. Table 3 shows how we can observe technical debt on a practical level (e.g. code, design and architecture) and how we can map these observations to different forms of technical debt (strategic, tactical, etc.). Hence, these associations are indeed speculative, being based on qualitative data from interviews, multivocal sources, and our overall understanding from the information that we collected.

Table 3 shows that an instance of technical debt can be classified in more than one form. According to the interview participants, design and architecture debt can manifest in all four forms of technical debt identified in the proposed taxonomy (Table 3). For example, a decision could be made to proceed with a design that creates a minimum viable product to access the market before implementing a stable rewrite – this is a form of strategic technical debt. Examples of tactical debt involving design and architecture include reactive decisions to defer the implementation of new technologies across all business functions, or rapidly coding a module for a solution without regard for the original design intent in order to meet a release deadline. Piecemeal design, particularly when executed at the level of an individual developer and not tracked, is a good example of incremental debt. Finally, design debt can also accrue inadvertently – for example, an incorrect assumption might be used during upfront design, or developers may not realise that they are not adhering to good design principles and the wider system architecture.

As code debt is accrued in small units at the level of the individual developer – e.g. with each duplicated block of code or each code comment that diverges from the actual implementation – it is most often an aspect of incremental or inadvertent technical debt. Whilst an application's code may be a component of strategic or tactical debt, the design of the coded solution, rather than the code itself is likely to constitute the debt in these situations.

The form of technical debt accrued by the deferral of upgrades and automations in an application's environment depends on the nature of the decision. For major upgrade and automation deferrals, strategic debt may be accrued as an application is nearing the end of its service life, and tactical debt is accrued when other tasks take priority. However, for minor issues that may appear too trivial to be tracked, and where decisions are often made in isolation without being communicated across the team, inadvertent debt results. Of course, environmental debt can also be inadvertent, when a conscious decision has not been made.

Technical debt related to knowledge distribution can be strategic as with the example of an application nearing the end of its service life, tactical when time constraints require the deferral

Table 3
Technical debt taxonomy matrix with its dimensions, attributes, precedents and outcomes.

	Technical debt taxonomy			
	Strategic	Tactical	Incremental	Inadvertent
Dimensions				
Code	–	–	•	•
Design & architecture	•	•	•	•
Environment	•	•	•	•
Hardware, infrastructure & supporting applications	•	•	•	•
Operational processes	•	•	•	•
Knowledge distribution	•	•	•	•
Documentation	•	•	•	•
Testing	•	•	•	•
Attributes				
Monetary cost	•	•	•	•
Difficulty to quantify	•	•	•	•
Amnesty	•	•	•	•
Bankruptcy	–	•	•	•
Interest & principal	•	•	•	•
Interest rate	•	•	•	•
Leverage	•	•	–	–
Debt ratio	•	•	–	–
Payoff line	•	•	–	–
Repayment & withdrawal	•	•	•	•
Credit rating	•	•	•	–
Low visibility	–	–	•	•
Precedents				
Pragmatism	•	•	•	–
Minimum viable product	•	•	•	–
Prioritisation	•	•	•	–
Time constraints	•	•	•	–
Budget constraints	•	•	–	–
Resource constraints	•	•	–	–
Customer & management expectations	•	•	–	–
Processes	–	•	•	–
Communication & collaboration	–	•	•	–
Lack of reviews	–	–	•	–
Attitudes	•	•	•	•
Apathy	–	•	•	•
Risk appetite	•	•	–	–
Ignorance	–	–	–	•
Oversight	–	–	–	•
Outcomes				
Morale	•	•	•	–
Productivity	•	•	•	–
Velocity	•	•	•	–
Regressions	–	–	–	–
Quality	•	•	•	•
Defects	•	•	•	•
Structural quality	•	•	•	•
Risk	•	•	•	•

of tacit or explicit knowledge distribution, incremental when many seemingly trivial decisions to defer documentation or tacit knowledge-sharing are made on an individual level, and inadvertent when it arises without any conscious decision-making

Risk-based testing is an example of strategic debt when it is proactively chosen due to the impracticality of executing all possible tests. A decision to defer testing tasks for a component in response to time constraints is an example of tactical debt. Incremental technical debt arises when testing deferrals are numerous, disparate and small such that they cannot be easily identified and tracked as a larger component. Finally, inadvertent technical debt arises when the absence of tests or automation of tests is not deliberate.

Fig. 2 summarises the theoretical framework developed from the research. As a conceptual device, the framework can be leveraged to inform action in response to perceived technical debt, and as a comprehensive guide when assessing software development practices. As a communication device, the framework can be used to aid developers in flagging issues that are rarely visible or considered at a management level.

A key outcome of this research is the creation of a proposed taxonomy (Table 3) that describes and encompasses different forms of the technical debt phenomenon and a theoretical framework (Fig. 2) for the phenomenon that can be thought to consist of technical debt's dimensions, attributes, precedents, and outcomes.

The taxonomy provides a useful structure that explores the relationships between each of these components. That is, not every instance of technical debt involves all the dimensions, attributes, precedents and outcomes described in the results section of this paper. The proposed taxonomy maps precedents and dimensions to different attributes and outcomes of technical debt, such that, given the reasons with which technical debt has been accrued (e.g. pragmatism), it is possible to anticipate likely attributes of the debt, which in turn lead to expectations of the outcomes (e.g., increased risk or decreased quality).

The theoretical framework aims to provide a comprehensive depiction of the technical debt phenomenon. As such, a practical approach should be taken when applying the framework in practice. As one interview participant notes, "*I think really keeping [the concept of technical debt] concentrated on the state of the codebase*

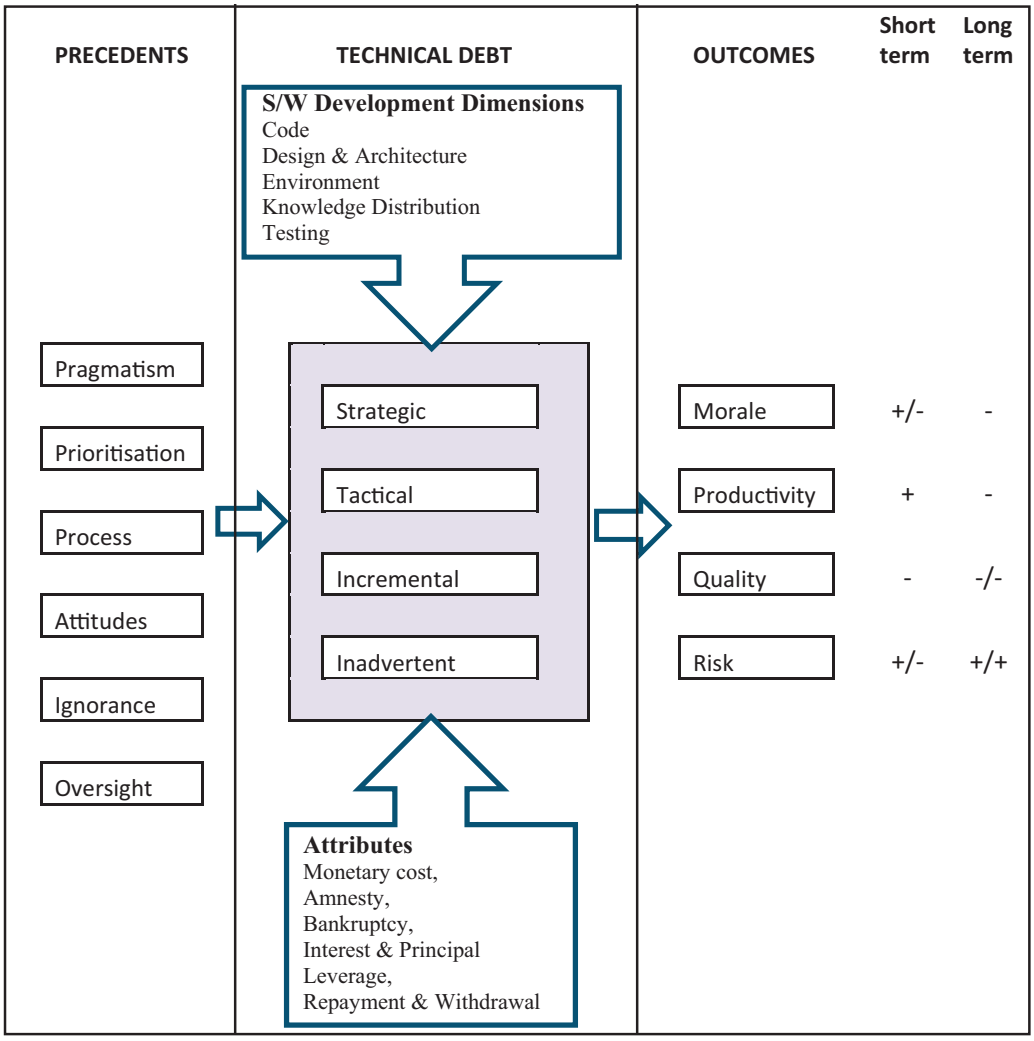

Fig. 2. Theoretical framework of technical debt.

you're working on is probably more useful than spreading it out to other disciplines and things that can involve other parts of the project that aren't really development".

The findings show that some overlap may exist between the precedents of technical debt – for example, there may be similar drivers of prioritisation and pragmatism. However this distinction is not pertinent to the utility of the theoretical framework outlined in this paper as our aim is to identify precedents rather than to study their interaction. Research findings also indicate that when debt is inadvertent – arising from an attitude of carelessness, or ignorance and oversight – it does not provide a form of leverage, unlike technical debt driven by other factors. Additionally, the impacts of inadvertent debt on productivity and morale typically take a while to manifest.

Teams and individuals should evaluate each dimension of technical debt on a case-by-case basis when using the framework to track their debt in a project. Whilst it may be helpful to recognise that technical debt can exist in areas external to a team or individual's immediate responsibilities, it may in fact be counterproductive to attempt to track or manage debt in such areas. For example, it might be more relevant for managers, rather than individual developers, to consider knowledge distribution as a form of technical debt. Similarly, it may not be necessary for practitioners to manage their technical debt giving consideration to all identified attributes of the phenomenon. Attributes such as the potential for technical debt amnesty may not be a relevant consideration for individual developers. Additionally, although attempting to associate a monetary cost with each instance of technical debt may be useful when communicating with management; this may not be the most suitable metric for tracking technical debt within teams.

5.4. Why teams should not strive for zero debt

Research findings indicate that the goal for practitioners shouldn't be to completely repay and avoid technical debt, but rather to focus on its adequate management. There are two principal reasons for this.

Firstly, research revealed a collective sentiment across academic and practitioner paradigms that the accrual of some technical debt is inevitable. As Shull (2011) notes, we find ourselves in a world of finite resources where prioritisation and trade-offs are constantly necessary. With regards to inadvertent technical debt, an interview participant notes that "*getting hung up over the fact that you do accumulate accidental debt is pointless – you can hire better people and get less of it, but you're never going to get rid of it*". Secondly, research revealed support for attributes of technical debt such as amnesty and leverage. These attributes suggest positive outcomes of technical debt when accrued carefully and in a controlled manner. Experiences shared by interview participants confirmed one of the fundamental principles of technical debt – that by taking out debt it is possible to accomplish something that was otherwise unattainable.

5.5. Research and practical implications

This research contributes fundamental theory that addresses a current gap in understanding, where knowledge of technical debt is abstract, high-level and anecdotal. The theoretical framework, including the taxonomy can inform and enable empirical studies investigating technical debt to be conducted, which in turn will further benefit both the academic and practitioner communities. The framework:

(i) helps practitioners to realise the utility of technical debt as a tool for conceptualisation and communication. It presents the technical debt concept in a more rigorous and comprehensive form;
(ii) facilitates more effective identification and acknowledgement of technical debt by highlighting aspects of software development where the potential for technical debt's existence might have been overlooked by individuals and teams;
(iii) can assist in making technical debt visible and accounted for, by allowing developers to more effectively communicate technical problems to management;
(iv) can help senior managers to think about software quality in business terms whilst managing software changeability and maintainability of their application portfolios;
(v) provides a taxonomy (Table 3) that promotes a holistic perspective on technical problems – from precedents, to attributes such as compounding interest payments and both short and long term outcomes. In doing so, the framework is able to serve as a guide for critically examining and enhancing software development processes with regards to the effects of technical debt.

This study has a number of limitations that need to be taken into account. Firstly, at the time this study was conducted the papers from the "Managing Technical Debt" workshop (ICSE 2010, 2011, 2012) and IEEE Software Special Issue on Technical Debt (Nov/Dec 2012) were not available. Hence, the papers from this workshop were not included in the systematic literature review. However, we have conducted a review of these papers and have included them as additional resources as appropriate. The inclusion of these resources has not changed the substantive content of the current study. Secondly, there were constraints on the search strategy for the MLR that was conducted. These constraints effectively limited the sampling of multivocal literature on technical debt to a subset such that only documents that were available online were considered, and each search iteration examined the first 50 most relevant results as determined by Google's search engine. The first constraint is unlikely to significantly impact results of the MLR since technical debt is a contemporary phenomenon, and the software development industry is one that is competent with using the World Wide Web as an information-sharing medium. The second constraint is acknowledged to be a limitation of the study, which supplementary interviews for data collection and informant reviews for evaluation attempt to address. Additionally, there is the risk that the phenomenon described by technical debt might be described with different terminology that is not covered by the search terms of this review. Whilst this risk was addressed by incorporating a scoping review and forward search into this review, it remains as an acknowledged validity threat.

Further, this study needs to acknowledge that the taxonomy in Table 3 and the theoretical framework in Fig. 2 is generated based on the MLR and the interviews. We believe that both Table 3 and Fig. 2 can be developed and tested through further studies – qualitative and quantitative (the later involving establishing of hypotheses, development of measurement models and testing the hypotheses). This taxonomy must be viewed with caution as it represents a qualitative analysis based on limited data.

6. Conclusion

This study focused on establishing a fundamental theoretical framework of technical debt organised around three research questions. The findings show that current thinking about technical debt is abstract, high-level and anecdotal. In addition to this the following points are illustrated:

- the phenomenon of technical debt, despite being described by a financial *metaphor*, is also associated with an *actual* monetary cost. Technical debt also exhibits many attributes of financial debt – it can be seen to have associated interest and principal, repayments and withdrawals, can be used for leverage, and, when left to accrue, technical debt can ultimately lead to amnesty or bankruptcy;
- precedents of technical debt include pragmatism, prioritisation, attitudes, ignorance and oversight. These precedents are not mutually exclusive and would be expected to manifest in various combinations and weights in different situations;
- morale, productivity, quality and risk are perceived to be fundamentally impacted by all forms of technical debt and to be particularly deleterious in the long term. Typically, regressions and defects appear in the long term as the result of increased complexity. Inadvertent technical debt does not affect productivity in the short term as it typically results in neither an increase or decrease in velocity, whilst morale is unaffected in the short term as inadvertent technical debt might not be discovered and realised for several iterations or more.

This study also provides a theoretical framework that incorporates the different dimensions, attributes, precedents and outcomes of technical debt. The proposed framework could perhaps explain the seemingly disparate themes surrounding technical debt in the literature, where there is a high degree of uncertainty and ambiguity regarding aspects of the phenomenon. This may be due to a general lack of exploration and understanding surrounding technical debt. However, it is possible that the lack of consideration of the need for taxonomy could be a barrier to establishing an effective understanding of the phenomenon. In summary, our framework (Fig. 2) and taxonomy (Table 3) provide a useful approach to understanding the overall phenomenon of technical debt for practical purposes. Future research should focus on how to turn the metaphor into literal "facts" that are used day-to-day to manage technical debt in the software domain.

Although the theoretical framework of this research was evaluated by the informants of the research, the next step for future research would be more rigorous validation and testing of the proposed framework. It would be beneficial for future research to focus on further qualifying these associations for different forms of technical debt and to seek to establish metrics to quantify technical debt and its impacts. Future research should also incorporate empirical studies to validate heuristics and techniques that will assist practitioners in their management of technical debt.

Appendix A. Multivocal sources

Iteration	Stage excluded	Source title	Source URL
1	3	Technical debt – Wikipedia, the free encyclopedia	http://en.wikipedia.org/wiki/Technical_debt
1	–	TechnicalDebt	http://martinfowler.com/bliki/TechnicalDebt.html
1	–	TechnicalDebtQuadrant	http://martinfowler.com/bliki/TechnicalDebtQuadrant.html
1	–	Design Debt – James Shore: The Art of Agile	http://jamesshore.com/Articles/Business/Software%20Profitability%20Newsletter/Design%20Debt.html
1	–	James Shore: Voluntary Technical Debt	http://jamesshore.com/Blog/CardMeeting/Voluntary-Technical-Debt.html
1	–	Coding Horror: Paying Down Your Technical Debt	http://www.codinghorror.com/blog/2009/02/paying-down-your-technical-debt.html
1	3	Design Debt	http://c2.com/cgi/wiki?DesignDebt
1	3	Technical Debt	http://c2.com/cgi/wiki?TechnicalDebt
1	–	Ward Explains Debt Metaphor	http://c2.com/cgi/wiki?WardExplainsDebtMetaphor
1	–	Technical Debt – 10x Software Development	http://forums.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx
1	2	Debt Metaphor – YouTube	http://www.youtube.com/watch?v=pqeJFYwnkjE
1	2	Technical Debt » a perspective on agile development, and other . . .	http://technicaldebt.com/
1	3	High Availability MySQL: Technical debt	http://mysqlha.blogspot.com/2011/08/technical-debt.html
1	–	When To Work On Technical Debt » Absolutely No Machete Juggling	http://www.nomachetejuggling.com/2011/07/22/when-to-work-on-technical-debt/
1	–	Technical Debt Estimation – CAST	http://www.castsoftware.com/Resources/Technical-Debt-Estimation.aspx
1	–	Product design debt versus Technical debt | Andrew Chen (@andrewchen)	http://andrewchenblog.com/2009/11/25/product-design-debt-versus-technical-debt/
1	3	InfoQ: Is Technical Debt Still a Useful Metaphor?	http://www.infoq.com/news/2011/07/death_of_tech_debt
1	3	InfoQ: Monetizing the Technical Debt	http://www.infoq.com/news/2010/03/monetizing-technical-debt
1	2	Technical Debt Plugin – Sonar – Codehaus	http://docs.codehaus.org/display/SONAR/Technical+Debt+Plugin
1	–	Lessons Learned: Embrace technical debt	http://www.startuplessonslearned.com/2009/07/embrace-technical-debt.html
1	–	Extending the Technical Debt Metaphor | Basildon Coder	http://basildoncoder.com/blog/2008/02/21/extending-the-technical-debt-metaphor/
1	–	Technical Debt and the Death of Design: Part 1 « Kane Mar	http://kanemar.com/2006/07/23/technical-debt-and-the-death-of-design-part-1/
1	3	Get out of technical debt: 1 of 72	http://petdance.com/perl/technical-debt/
1	3	Second International Workshop on Managing Technical Debt	http://www.sei.cmu.edu/community/td2011/
1	–	Technical Debt – Andrew Stopford's Weblog	http://weblogs.asp.net/astopford/archive/2010/07/19/technical-debt.aspx
1	–	Design debt economics: A vocabulary for describing the causes . . .	http://www.ibm.com/developerworks/rational/library/edge/09/jun09/designdebteconomics/
1	–	Avoid getting buried in technical debt | TechRepublic	http://www.techrepublic.com/blog/programming-and-development/avoid-getting-buried-in-technical-debt/3849
1	–	Going Into Design Debt	http://haacked.com/archive/2005/09/24/GoingIntoDesignDebt.aspx
1	–	Technical Debt	http://thecriticalpath.info/2011/01/16/technical-debt/
1	3	Technical Debt	http://www.slideshare.net/garyshort/technical-debt-2985889
1	–	Paying Down Technical Debt | BrandonSavage.net	http://www.brandonsavage.net/paying-down-technical-debt/
1	3	Naresh Jain » Agile FAQs Blog » Managed Chaos » Technical Debt	http://blogs.agilefaqs.com/2011/02/16/technical-debt/
1	–	Code Cleanup: Using Agile Techniques to Pay Back Technical Debt	http://msdn.microsoft.com/en-us/magazine/ee819135.aspx
1	–	Powers of Two: Ward Cunningham's Debt Metaphor Isn't a Metaphor	http://powersoftwo.agileinstitute.com/2009/03/ward-cunninghams-debt-metaphor-isnt.html
1	–	Technical Debt – Pat Maddox, B.D.D.M.F.	http://patmaddox.com/blog/technical-debt.html
1	–	Technical Debt – Should you avoid it? – dopefly dot com – The . . .	http://www.dopefly.com/techblog/entry.cfm?entry=321
1	–	Seven Strategies for Technical Debt	http://www.theagileengineer.com/public/Home/Home_files/TechnicalDebt_published.pdf
1	–	Jay Fields' Thoughts: Types of Technical Debt	http://blog.jayfields.com/2011/03/types-of-technical-debt.html
1	–	Why Technical Debt Matters | NetObjectives	http://www.netobjectives.com/blogs/why-techncial-debt-matters
1	3	Talk Like A Duck: Ward Cunningham on the Debt Metaphor	http://talklikeaduck.denhaven2.com/2009/03/02/ward-cunningham-on-the-debt-metaphor

Appendix A. (*Continued*)

Iteration	Stage excluded	Source title	Source URL
1	–	ACCU:: Managing Technical Debt	http://accu.org/index.php/journals/1301
1	2	Design Debt	http://architecturalpractices.com/articles_design_debt.html
1	–	Sonar » Evaluate your technical debt with Sonar	http://www.sonarsource.org/evaluate-your-technical-debt-with-sonar/
1	2	Types of Technical Debt | Agile Zone	http://agile.dzone.com/articles/types-technical-debt
1	–	Bad code isn't Technical Debt, it's an unhedged Call Option . . .	http://www.higherorderlogic.com/2010/07/bad-code-isnt-technical-debt-its-an-unhedged-call-option/
1	3	Agile Testing with Lisa Crispin » Blog Archive » Technical Debt . . .	http://lisacrispin.com/wordpress/2011/06/22/technical-debt-sprints/
1	–	The Hidden Costs of Technical Debt – Agile Web Development . . .	http://www.agileweboperations.com/the-hidden-costs-of-technical-debt
1	–	Know your Technical Debt Burden by AntiClue	http://www.anticlue.net/archives/IT-Leadership/TechnicalDebtBurden.htm
1	–	Quantifying Technical Debt « Arlo Being Bloody Stupid	http://arlobelshee.com/essay/quantifying-technical-debt
1	3	Technical Debt: Assessment and Reduction « The Agile Executive	http://theagileexecutive.com/2011/08/05/technical-debt-assessment-and-reduction/
2	1	Technical debt – Wikipedia, the free encyclopedia	http://en.wikipedia.org/wiki/Technical_debt
2	3	Code Debt | BrainFuel	http://www.brainfuel.tv/code-debt
2	3	Getting Real: Manage Debt (by 37signals)	http://gettingreal.37signals.com/ch10_Manage_Debt.php
2	3	Article info: Staying Out of Code Debt	http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=9860&tth=DYN&tt=siteemail&iDyn=2
2	1	Quality Debt Management Services and Consumer Credit Resources	http://www.qualitydebtmanagementservices.com/
2	–	Code is Money: Code Debt	http://www.search-this.com/2007/03/13/code-is-money-code-debt/
2	1	Coding Horror: Paying Down Your Technical Debt	http://www.codinghorror.com/blog/2009/02/paying-down-your-technical-debt.html
2	1	Technical Debt	http://c2.com/cgi/wiki?TechnicalDebt
2	1	Poll: Conservatives Oppose Changing Tax Code, Debt Ceiling Hike . . .	http://www.prnewswire.com/news-releases/poll-conservatives-oppose-changing-tax-code-debt-ceiling-hike-126259708.html
2	3	On WM 2.3 and Code Debt | World Machine Development Diary	http://world-machine.com/blog/?p=229
2	1	Silver: Quality Silver Bullion July Discount Code, Debt Ceiling and . . .	http://www.youtube.com/watch?v=pQkzbK4NsxA
2	1	Bimo Marketing Quality Debt Leads – YouTube	http://www.youtube.com/watch?v=AE8AGEE43Qg
2	3	Code debt | Facebook	http://www.facebook.com/pages/Code-debt/103259639727447
2	1	Dailymotion – High Quality Debt Leads Exchange | DebtLeadsExchange . . .	http://www.dailymotion.com/video/xfnadw_high-quality-debt-leads-exchange-debtleadsexchange-com_shortfilms
2	3	Urban Dictionary: code debt	http://www.urbandictionary.com/define.php?term=code%20debt
2	3	Talking about code debt. | Moritz Haarmann's Blog	http://momo.brauchtman.net/2011/08/15/talking-about-code-debt/
2	3	Code Debt: Neither A Borrower. . .	http://tynerblain.com/blog/2007/01/12/code-debt/
2	2	code debt	http://tynerblain.com/blog/tag/code-debt/
2	1	[WTS] 5 High Quality Debt Settlement Articles	http://forums.digitalpoint.com/showthread.php?t=1890540
2	1	Debt Leads | Debt Lead | DebtLeadsExchange.com	http://debtleadsexchange.com/
2	1	Poll: Conservatives Oppose Changing Tax Code, Debt Ceiling Hike	http://www.istockanalyst.com/business/news/5318899/poll-conservatives-oppose-changing-tax-code-debt-ceiling-hike

2	1	Where can i get quality debt consolidation information? – Yahoo . . .	http://answers.yahoo.com/question/index?qid=20110330063850AANKQvB
2	1	Debt Manager Professional Free Download at datapicks.com – Create . . .	http://debt-manager-professional.download-486-30100.datapicks.com/
2	1	High Quality Debt Management Plans – Simplest Way To Get Debt Free . . .	http://www.investmentfinancialadvice.com/2011/05/27/high-quality-debt-management-plans-simplest-way-to-get-debt-free/
2	1	Quality Debt Settlement Businesses | Vision Debt Solutions	http://www.visiondebt.com/quality-debt-settlement-businesses/
2	1	Quality Debt Settlement Leads and Live Transfer Debt Leads	http://www.xleadsinc.com/debt-settlement.php
2	1	I am looking to generate quality debt leads via the CPL Model – Not . . .	http://www.linkedin.com/groups/I-am-looking-generate-quality-1857863.S.44108793
2	1	Quality Debt, Mortgage, Real Estate Leads | LinkedIn	http://www.linkedin.com/company/bimo-marketing/quality-debt-mortgage-real-estate-leads-180801/product
2	3	Code Debt-Free | End Point Blog	http://blog.endpoint.com/2008/07/code-debt-free.html
2	1	Professional Credit Repair Software Create high quality debt . . .	http://debt-settlement.qarchive.org/
2	–	Digerati Illuminatus: More on Code Debt	http://digerati-illuminatus.blogspot.com/2008/03/more-on-code-debt.html
2	–	Digerati Illuminatus: Code Debt, Product Market Debt, and Customer . . .	http://digerati-illuminatus.blogspot.com/2007/11/code-debt-product-market-debt-and.html
2	1	Poll: Conservatives Oppose Changing Tax Code, Debt Ceiling Hike	http://www.mbsmagazine.com/world-a-regional/usa/820-debt-ceiling-hike.html
2	1	$$$ Quality Debt Sett. & Payday Loan Leads . . . – Freelancer.com	http://www.freelancer.com/projects/Internet-Marketing-Telemarketing/Quality-Debt-Sett-amp-Payday.html
2	3	Staying Out of Code Debt | Agile	http://agile.techwell.com/articles/membersub/staying-out-code-debt
2	1	Quality Debt Support Service – Debt Help Services	https://sites.google.com/site/debthelp01/quality-debt-support-service
2	1	Get Out Of Debt – Create high quality debt management plans for . . .	http://get-out-of-debt.softalizer.com/
2	1	Debt Management Software – Create high quality debt management . . .	http://debt-management-software.softalizer.com/
2	1	Debt Manager Professional 4.0 Free Download. Create high quality . . .	http://debt-manager-professional.lastdownload.com/
2	1	Personal debt – Plan to eliminate your debt now. Create high . . .	http://personal-debt.safe-install.com/
2	3	Technical debt – what about data quality debt? « Michael Baylon's blog	http://michaelbaylon.wordpress.com/2009/12/29/technical-debt-what-about-information-quality-debt/
2	1	Uniform Commercial Code Debt Collection – Portal for . . .	http://www.idi.ntnu.no/datamuseum/Members/sheriecortez/uniform-commercial-code-debt-collection
2	1	What Produces an Excellent High Quality Debt Lead?	http://www.squidoo.com/what-produces-an-excellent-high-quality-debt-lead-
2	1	Hungry For Good-quality Debt	http://www.tremors.com.au/tremors-articles/2008/2/13/hungry-for-goodquality-debt/
2	1	Quality debt management.com limited	http://www.cdrex.com/quality-debt-managementcom-limited-6316363.html
2	1	Poor quality debt	http://www.proz.com/kudoz/english_to_german/finance_general/1937676-poor_quality_debt.html
2	1	Best Quality Debt Leads Exchange High Quality Debt . . . – Metacafe	http://www.metacafe.com/watch/5553413/best_quality_debt_leads_exchange_high_quality_debt_leads_debtle/
2	1	Quality Debt Improvement and Repair	http://debtandcredithelp.org/
2	1	Quality debt lead source wanted	http://www.affiliateseeking.com/forums/general-marketing/2834-quality-debt-lead-source-wanted.html
2	1	Quality Debt Settlement Leads | Debt Leads | Tax Settlement Leads . . .	http://www.predictivedollars.com/

Appendix B. Extracted data and themes

Themes	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16	A17	A18	A19
Code (RQ1)	X	X	X			X	X		X	X						X	X		
Design / Architecture (RQ1)	X	X	X	X		X	X					X			X		X	X	
Defects / Issues (RQ1)			X				X										X		
Documentation (RQ1)	X		X			X													
Infrastructure (RQ1)											X								
Testing (RQ1)	X		X			X													
Unimplemented Features (RQ1)			X							X									
Bankruptcy (RQ1, RQ3)						X													
Interest (RQ1, RQ3)	X		X							X		X	X		X		X	X	
Relationship to Quality (RQ1, RQ3)		X	X	X						X		X				X			
Short / Long Term (RQ3)					X					X									
Difficulty and Risk (RQ3)		X	X	X	X	X	X		X		X			X		X	X		
Healthy vs. Unhealthy (RQ2, RQ3)	X				X														X
Quadrant (RQ2, RQ3)			X			X													
Timelines / Scheduling / Velocity (RQ2, RQ3)	X	X	X	X		X	X	X		X		X	X		X			X	X
Budget / Resource Constraints (RQ2)						X				X									
Visibility of Debt (RQ2)				X	X			X		X									
Type of Study	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**	**A11**	**A12**	**A13**	**A14**	**A15**	**A16**	**A17**	**A18**	**A19**
Expert Opinion / Secondary Study	X		X	X		X	X	X		X				X	X	X	X	X	
Primary Study or Experience Report		X			X				X		X	X	X						X
Primary Study examining Technical Debt																			
Definition or Conceptual Model	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**	**A11**	**A12**	**A13**	**A14**	**A15**	**A16**	**A17**	**A18**	**A19**
Comprehensive / Complete																			
Incomprehensive / Incomplete	X	X	X	X	X	X	X			X		X	X		X	X	X		X
Negligible or Absent								X	X		X			X				X	

Appendix C. Explicit and implied definitions of technical debt in academic literature

Definition	Source
"Almost invariably in software projects, developers can be so focused on accomplishing the needed functionality that the software itself grows less understandable, more complex, and harder to modify"	A1 (Shull, 2011)
"This pressure [to meet deadlines] *can encourage shortcuts concerning code maintenance that lead to the accumulation of technical debt, that is, a backlog of deferred technical problems"*	A2 (Torkar et al., 2011)
"Cunningham coined the technical debt metaphor in his 1992 OOPSLA experience report to describe a situation in which long-term code quality is traded for short-term gain, creating future pressure to remediate the expedient"	A3 (Brown et al., 2010b)
"This [simplistically allowing business value to drive development so that architectural soundness is compromised] *may lead to increasing maintenance costs and the quality of the end product is undermined"*	A4 (Heidenberg and Porres, 2010)
"As defined by David Draper, at its broadest, technical debt is any side of the current system that is considered sub-optimal from a technical perspective"	A5 (Ktata and Lévesque, 2010)
"The technical debt metaphor was first introduced by Ward Cunningham to compare deficiencies in software with financial debt"	A6 (O'Connor, 2010)
"As they deliver software, teams accrue what the agile community refers to as 'technical debt' in their code. This includes things like bugs, design issues, and other code-quality problems that are potentially introduced with every addition or change to the code"	A7 (Black et al., 2009)
"The technical debt present is a byproduct of the previous private loan project, as most business and technical decisions were prioritised by the business team"	A8 (Davis and Andersen, 2009)
"Some industry experts view poorly evolvable code as technical debt that can slow down development"	A9 (Mantyla and Lassenius, 2009)
"If teams are making decisions to sacrifice quality or maintainability in order to meet those demands [pressures to use fewer resources, hit timelines and show return on investment], *technical debt is incurred"*	A10 (Smith, 2009)
"Non-agile infrastructure tends to grow old because changes are hard to execute in these environments. This can be seen as technical debt"	A11 (Debois, 2008)
"Changing priority to short-term aspects usually contributes to increased technical debt and decreased project quality"	A12 (Lindgren et al., 2008b)
"Technical debt refers to 'software aging' costs that are not attended to, which hence need to be repaid at a later time"	A13 (Lindgren et al., 2008a)
"Ward Cunningham coined the term 'technical debt' to explain how we should manage this [attending to code quality and design maintenance] *effort"*	A14 (Wirfs-Brock, 2008a)
"I'm acutely aware that when I let my design slide, I'm creating technical debt"	A15 (Wirfs-Brock, 2008b)
"Complex software systems erode over time. Software systems must be extended, adapted, and modified accordingly as new requirements, constraints, and environments emerge. Developers, however, seldom give these efforts the rigorous consideration of the original design. Consequently, the system decays, resulting in decreased usefulness and increased errors"	A16 (Neill and Laplante, 2006)
"Incomplete changes are introduced into the system, each incurring a 'technical debt' that represents the degree of incompleteness"	A17 (Klein, 2005)
"Just as new firms borrow capital to get started, new software projects borrow 'design capital' (time) to get a product to market. Maintenance problems that ensue are the interest you pay for design errors introduced by schedule pressure"	A18 (Lutz, 1993)
"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"	A19 (Cunningham, 1992)

Appendix D. Interview schedule

Opening Questions	
1. What is your current position or role?	Background
2. What have been the types of responsibilities of the positions you've held that relate to software development?	
3. Approximately how long have you been doing work related to software development?	
4. Could you briefly describe your experiences related to the industry, including the types of methods and processes you have used?	
5. Are you familiar with the concept of technical debt?	Understanding
IF YES -	
a. How would you describe what technical debt is?	
IF NO -	
b. Just from hearing the phrase, what do you think technical debt might mean?	
c. The technical debt concept basically applies financial debt as a metaphor for what happens when changes to software are more costly to make in the future.	
Provide this definition on a slip of paper that the interviewee can refer to throughout the interview.	
RQ1 – What are the dimensions of technical debt?	
6. Based on your experiences and your interpretation of what technical debt is, what do you think constitutes technical debt?	Dimensions – Attribute Elicitation
Ask follow-up questions to refine the granularity and detail of responses (including examples).	
If any of the following dimensions have not already been addressed by the interviewee, ask the corresponding question with follow-up questions to refine the granularity and detail of responses (including examples).	Dimensions – Attribute Confirmation
7. Code Decay – Do you think poor quality code constitutes technical debt? Examples might include unnecessary coupling, a lack of abstraction, code duplication, and quick rather than elegant solutions.	
8. Design/Architectural Debt – Do you think degraded system architecture from a lack of structural refactoring constitutes technical debt?	
9. Known Issues/Defects – Do you think known issues and defects that haven't been addressed constitute technical debt?	
10. Unimplemented Features – Do you features that have not yet been implemented constitute technical debt?	
11. Infrastructural Debt – Do you think postponed infrastructural changes and updates constitute technical debt?	
12. Documentation Debt – Do you think inadequate documentation constitutes technical debt?	
13. Testing Debt – Do you think inadequate testing constitutes technical debt?	
14. Can you think of examples where the analogies of interest and bankruptcy apply to technical debt in software development?	Metaphors
15.Other than interest and bankruptcy, can you think of any other aspects of the debt metaphor that are applicable to software development?	
RQ2 – How does technical debt arise?	
16. What do you think are the causes of technical debt?	Precedents – Attribute Elicitation
Ask follow-up questions to refine the granularity and detail of responses (including examples).	
If any of the following dimensions have not already been addressed by the interviewee, ask the corresponding question with follow-up questions to refine the granularity and detail of responses (including examples).	Precedents – Attribute Confirmation
17. Project Constraints – Do you think project constraints such as time, budgeting and resourcing cause technical debt?	
18. Low Visibility of Debt – Do you think the fact that technical debt isn't easily identified and communicated is a cause for its accrual?	
For both of the following questions – ask follow-up questions to refine the granularity and detail of responses (including examples).	Behaviours
19. How do you think a team being reckless or prudent affects technical debt in a project?	
20. Do you think it makes a difference if technical debt is deliberate or inadvertent?	
21.Do you think any other types of behaviours affect technical debt?	
RQ3 – What are the benefits and drawbacks of allowing technical debt to accrue?	
For both of the following questions – ask follow-up questions to refine the granularity and detail of responses (including examples).	Outcomes – Attribute Elicitation
22. What do you think are the drawbacks of allowing technical debt to accrue?	
23. What do you think are the benefits of allowing technical debt to accrue, if any?	
If any of the following dimensions have not already been addressed by the interviewee, ask the corresponding question with follow-up questions to refine the granularity and detail of responses (including examples).	Outcomes – Attribute Confirmation
24. Interest – Do you think technical debt becomes increasingly costly to repay as time passes?	
25. Bankruptcy – Do you think technical debt could lead to the situation where it's more costly to make a change to an existing system compared to completely rebuilding that system to incorporate the change?	
26. Scheduling/Timelines – Do you think technical debt impacts on scheduling and timelines?	
27. Difficulty/Risk – Do you think technical debt results in increased difficulties and risks in a software development project?	
28. Quality – Do you think technical debt affects the quality of software?	
29. Do you think there are healthy and unhealthy forms of technical debt? If so – what would you consider to be healthy and unhealthy forms of debt?	Healthy vs. Unhealthy
Ask follow-up questions to refine the granularity and detail of responses, and also to determine how healthy and unhealthy classifications relate to technical debt outcomes.	
Closing Questions	
30. Do you know of any other concepts, models, or paradigms related to technical debt?	Additional Information
31.Is there anything else you'd like to share about your experiences with software development?	

References

Allman, E., 2012. Managing technical debt. ACM 5 (May (5)), 50–55.

Atwood, J., 2009. Paying down your technical debt. In: Coding Horror, Available from: http://www.codinghorror.com/blog/2009/02/paying-down-your-technical-debt.html (accessed 24.08.11). (Online).

Black, S., Boca, P.P., Bowen, J.P., Gorman, J., Hincley, M., 2009. Formal versus agile: survival of the fittest. Computer 42, 37–45, *A8.

Brazier, T., 2007. Managing technical debt. ACCU, Available from: http://accu.org/index.php/journals/1301 (accessed 24.08.11). (Online).

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., Maccormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N., 2010a. Managing technical debt in software-reliant systems. FSE/SDP Workshop on the Future of Software Engineering Research, FoSER 2010, Santa Fe, NM, United states, November 7, 2010–November 11, 2010. Association for Computing Machinery, 47–51, *A3.

Brown, N., Nord, R., Ozkaya, I., 2010b. Enabling Agility through Architecture. CrossTalk, pp. 12–17.

Camden, C., 2011. Avoid Getting Buried in Technical Debt, Available from: http://www.techrepublic.com/blog/programming-and-development/avoid-getting-buried-in-technical-debt/3849 (accessed 24.08.11).

Cast, 2011. Reduce Technical Debt, Available from: http://www.castsoftware.com/solutions/reduce-technical-debt (accessed 19.10.11). (Online).

Cunningham, W., 1992. The WyCash portfolio management system. Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum), Vancouver, British Columbia, Canada. ACM, *A19.

Davis, J.D., Andersen, T.J., 2009. Surviving the economic downturn. 2009 Agile Conference, AGILE 2009, Chicago, IL, United states, August 24, 2009–August 28, 2009. IEEE Computer Society, 245–250.

Debois, P., 2008. Agile infrastructure and operations: how infra-gile are you? In: Agile, 2008. AGILE '08. Conference, 4–8 August 2008, pp. 202–207, *A11.

Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A., 2001. Does Code Decay? Assessing the Evidence from Change Management Data. IEEE Transactions on Software Engineering 27 (1), 1–12.

ELM, J., 2009. Design Debt Economics: A Vocabulary for Describing the Causes, Costs and Cures for Software Maintainability Problems. IBM, Available: http://www.ibm.com/developerworks/rational/library/edge/09/jun09/designdebteconomics/ (accessed 24.08.11). (Online).

Fields, J., 2011. Types of Technical Debt, Available from: http://www.blog.jayfields.com/2011/03/types-of-technical-debt.html (accessed 24.08.11).

Fowler, M., 2009a. TechnicalDebt, Available from: http://www.martinfowler.com/bliki/technicaldebt.html (accessed 28.04.11).

Fowler, M., 2009b. TechnicalDebtQuadrant, Available from: http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html (accessed 28.04.11).

Guo, Y., Seaman, C., 2011. A portfolio approach to technical debt. In: Second International Workshop on Managing Technical Debt, ICSE 2011, Waikiki, Honolulu, Hawaii, USA, 23 May 2011.

Haack, P., 2005. Going Into Design Debt, Available from: http://haacked.com/archive/2005/09/24/GoingIntoDesignDebt.aspx (accessed 24.08.11).

Heidenberg, J., Porres, I., 2010. Metrics functions for Kanban guards. In: 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS), 2010, 22–26 March 2010, pp. 306–310, *A4.

Hilton, R., 2011. When to work on technical debt. In: Absolutely No Machete Juggling, Available from: http://www.nomachetejuggling.com/2011/07/22/when-to-work-on-technical-debt/ (accessed 24.08.11). (Online).

Kitchenham, B., 2004. Procedures for Performing Systematic Reviews. Keele University.

Klinger, T., Tarr, P., Wagstrom, P., Williams, C., 2011. An enterprise perspective on technical debt. In: Second International Workshop on Managing Technical Debt, ICSE 2011, Waikiki, Honolulu, Hawaii, USA, 23 May 2011.

Klein, J., 2005. How does the architect's role change as the software ages? In: 5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005, p. 141, *A17.

Ktata, O., Lévesque, G., 2010. Designing and implementing a measurement program for scrum teams: what do agile developers really need and want? 3rd C* Conference on Computer Science and Software Engineering 2010, C3S2E '10, Montreal, QC, Canada, May 19, 2010–May 20, 2010. Association for Computing Machinery, 101–107, *A5.

Laribee, D., 2009. Using agile techniques to pay back technical debt. MSDN Magazine, Available from: http://msdn.microsoft.com/en-us/magazine/ee819135.aspx (accessed 24.08.11). (Online).

Lim, E., Taksande, N., Seaman, C., 2012. A balancing act: what software practitioners have to say about technical debt. IEEE Software, 22–28.

Lindgren, M., Land, R., Norstrom, C., Wall, A., 2008a. Key aspects of software release planning in industry. In: 19th Australian Conference on Software Engineering, 2008. ASWEC 2008, 26–28 March 2008, pp. 320–329, *A13.

Lindgren, M., Wall, A., Land, R., Norstrom, C., 2008b. A method for balancing short- and long-term investments: quality vs. features. In: 34th Euromicro Conference on Software Engineering and Advanced Applications, 2008. SEAA'08, 3–5 September 2008, pp. 175–182, *A12.

Lutz, M.J., 1993. A maturing field on display at OOPSLA. IEEE Software 10, 113, *A18.

Mantyla, M.V., Lassenius, C., 2009. What types of defects are really discovered in code reviews? IEEE Transactions on Software Engineering 35, 430–448, *A9.

McConnell, 2007. Technical debt. In: 10x Software Development, Available from: http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx (accessed 28.04.11). (Online).

Miles, M.B., Huberman, A.M., 1994. Qualitative Data Analysis: An Expanded Sourcebook. Sage Publications, Thousand Oaks, CA.

Myers, R., 2009. Ward Cunningham's debt metaphor isn't a metaphor. In: Powers of Two, Available from: http://powersoftwo.agileinstitute.com/2009/03/ward-cunninghams-debt-metaphor-isnt.html (accessed 16.05.11). (Online).

Neill, C.J., Laplante, P.A., 2006. Paying down design debt with strategic refactoring. Computer 39, 131–134, *A16.

Nielsen, E., 2011. Know Your Technical Debt Burden, Available from: http://www.anticlue.net/archives/IT-Leadership/TechnicalDebtBurden.htm (accessed 24.08.11). (Online).

O'Connor, D., 2010. Technical debt in semiconductor equipment: It's time to pay it down. Solid State Technology 53, 34–35, *A6.

Ogawa, R.T., Malen, B., 1991. Towards rigor in reviews of multivocal literatures: applying the exploratory case study method. Review of Educational Research 61, 265–286.

Ries, E., 2009. Embrace technical debt. In: Lessons Learned, Available from: http://www.startuplessonslearned.com/2009/07/embrace-technical-debt.html (accessed 24.08.11). (Online).

Seaman, C., Guo, Y., Izurieta, C., Cai, Y., Zazwirja, N., Shull, F., Vetro, A., 2012. Using technical debt data in decision making: potential decision approaches. In: Second International Workshop on Managing Technical Debt, ICSE 2011, Waikiki, Honolulu, Hawaii, USA, 23 May 2011.

Shalloway, A., 2011. Why Technical Debt Matters, Available from: http://www.netobjectives.com/blogs/why-techncial-debt-matters (accessed 24.0811).

Shriver, R., 2010. Seven Strategies for Technical Debt, Available: http://www.theagileengineer.com/public/Home/Home_files/TechnicalDebt_published.pdf (accessed 24.08.11). (Online).

Shull, F., 2011. Perfectionists in a world of finite resources. IEEE Software 28, 4–6, *A1.

Silverman, D., 2000. Doing Qualitative Research. Sage Publications, London.

Slinker, G., 2007. Code debt, product market debt, and customer debt. In: Digerati Illuminatus, Available from: http://digerati-illuminatus.blogspot.com/2007/11/code-debt-product-market-debt-and.html (accessed 26.09.11). (Online).

Slinker, G., 2008. More on code debt. In: Digerati Illuminatus, Available from: http://digerati-illuminatus.blogspot.com/2008/03/more-on-code-debt.html (accessed 26.09.11). (Online).

Smith, R., 2009. Computer system design. Journal of GXP Compliance 13, 44–48, *A10.

Snipes, W., Robinson, B., Guo, Y., Seaman, C., 2012. Defining the decision factors for managing defects: a technical debt perspective. In: Third International Workshop on Managing Technical Debt, ICSE 2012, Zurich, Switzerland, June 5, 2012.

Szykarski, A., 2012. Ted Theodorpoulos on Managing Technical Debt Successfully, Available from: http://www.ontechnicaldebt.com/blog/ted-theodoropoulos-on-managing-technical-debt-successfully/

Stopford, A., 2010. Technical Debt, Available from: http://www.weblogs.asp.net/astopford/archive/2010/07/19/technical-debt.aspx (accessed 24.08.11).

Thibodeau, P., 2011. Counting 'technical debt'. In: Information Age, p. 64, *A7.

Tom, E., 2011. A Consolidated Understanding of Technical Debt. Honours Thesis. School of Information Systems, Technology and Management, University of New South Wales, Sydney, 2052, Australia.

Tom, E., Aurum, A., Vidgenr, 2012. A consolidated understanding of technical debt. In: European Conference on Information Systems (ECIS 2012), Barcelona, Spain, 10–13 June.

Torkar, R., Minoves, P., Garrigós, J., 2011. Adopting free/libre/open source software practices, techniques and methods for industrial use*. Journal of the Association for Information Systems 12, 88–122, *A2.

Wiklund, K., Eldh, S., Sundmark, D., Lundqvist, K., 2012. Technical debt in test automation. In: Third International Workshop on Managing Technical Debt, ICSE 2012, Zurich, Switzerland, June 5, 2012.

Wirfs-Brock, R., 2008a. Designing Then and Now. IEEE Software 25, 29, *A14.

Wirfs-Brock, R.J., 2008b. Enabling change. IEEE Software 25, 70–71, *A15.

Zazworka, N., 2011. Prioritizing design debt: investment opportunities. In: Second International Workshop on Managing Technical Debt, ICSE 2011, Waikiki, Honolulu, Hawaii, USA, 23 May, 2011.

Edith Tom works as a software developer at Atlassian. Her research interests include technical debt and how it is managed in software development. Edith has a B.Sc. in Business Information Technology from the University of New South Wales, Australia, and she is currently studying for a Masters of Information Technology at the University of New South Wales.

Aybüke Aurum is an associate professor at the School of Information Systems, Technology and Management, University of New South Wales, Australia. She received her Ph.D. in computer science. Her research interests include agile software

development, project management, value-based approach in software development, and requirements engineering. She is on the editorial boards of Information and Software Technology Journal and Requirements Engineering Journal. She is a member of IEEE and Australian Computer Society.

Richard Vidgen is Professor of Systems Thinking in the Hull University Business School. He worked for fifteen years in the IT industry as a programmer, systems analyst, and project manager. On leaving industry he studied for a PhD in Information Systems at the University of Salford and then worked at the School of Management, University of Bath and the Australian School of Business, University of New South Wales. He has published research articles in leading journals including Information Systems Research, Information & Management, European Journal of Information Systems, Journal of Information Technology, Omega, Journal of Strategic Information Systems, and the Information Systems Journal. His current research interests include: the application of complex adaptive systems theory and social networks to the study of information system and software development; Internet quality and Web site usability; and, the assessment of scholarly influence and journal quality.