Noname manuscript No.
(will be inserted by the editor)

Measuring affective states from technical debt

A psychoempirical software engineering experiment

Jesper Olsson · Erik Risfelt · Terese Besker · Antonio Martini · Richard Torkar

Received: date / Accepted: date

Abstract Software engineering is a human activity. Despite this, human aspects are under-represented in technical debt research, perhaps because they are challenging to valorize.

This study's objective was to investigate the relationship between technical debt and affective states (feelings, emotions, and moods) from software practitioners. Forty participants from twelve companies took part in a mixed-methods design, consisting of a repeated-measures experiment, a survey employing a questionnaire, and semi-structured interviews.

The statistical analysis shows that different design smells negatively or positively impact affective states. From the qualitative data, it is clear that technical debt activates a substantial portion of the emotional room and is psychologically taxing. Further, reactions to technical debt appear to fall in different levels of maturity.

J. Olsson, E. Risfelt, T. Besker
Chalmers and University of Gothenburg
Dept. of Computer Science and Engineering
SE-412 96 Göteborg, Sweden
E-mail: jesper.olsson1@saabgroup.com

R. Torkar
Chalmers and University of Gothenburg
Dept. of Computer Science and Engineering
SE-412 96 Göteborg, Sweden
Stellenbosch Institute for Advanced Study (STIAS)
Wallenberg Research Centre at Stellenbosch University
Stellenbosch, South Africa

A. Martini
University of Oslo
Dept. of Informatics
N-0373 Oslo, Norway

We argue that human aspects in software engineering are an essential factor to consider, as it can result in, e.g., procrastination, apprehension, and burnout.

1 Introduction

Software engineering is very much a human activity, but this is sometimes forgotten. When proposing hypotheses, analyzing results, and discussing implications for industry, we researchers sometimes neglect to factor in human aspects. So, too, is the case in technical debt research. This paper intends to amend this deficit and provide evidence showing that technical debt has noticeable adverse effects on software practitioners' feelings.

Technical Debt (TD) is a financial metaphor typically used within software engineering to explain long-term costs of short-term benefits. It is a communicative aid for bridging the knowledge gap between software practitioners and decision-makers. Hence, if the metaphor was to miscount (or not account for) pivotal cost-benefit factors, the effect could be detrimental to software companies. As alluded to in the previous paragraph, the fact of the matter is that TD's most recent definition is incomplete.

In the academic world, the current definition (i.e., the ISO 16162 definition of technical debt) was agreed upon during the Dagstuhl seminar in 2016 (Avgeriou et al., 2016). This definition is nuanced, incorporates decades of research, and offers a shared understanding of TD. Among many other things, it is emphasized that TD is a software development artifact in its own right and that TD acquisition is not necessarily intentional nor visible. A list of various consequences was also synthesized, but it falls short in recognizing the effects of TD on the human aspects of software engineering.

The goal of this paper is to fill that gap. By employing a mixed-methods approach (including experiments with human subjects) and following guidelines for psychoempirical software engineering research, this study collected empirical data (200 data points from $n = 40$ subjects) on how design debt influences the so-called affective state of software practitioners. Applying Bayesian multi-level models revealed, among other findings, strong evidence that certain design smells (notably cyclic-dependencies) caused the participants displeasure. The qualitative analysis suggests that many practitioners experience anxiety from high amounts of TD, and their responses vary along a maturity scale.

In more concrete terms, the research objective of this study is to investigate the relationship between TD and affective state from the point of view of software practitioners. The following research questions support the objective.

RQ1: How do software practitioners' affective state change in the presence of design smells?

RQ2: How do changes in affective state align with professional characteristics (e.g., formal education, work experience, or work context)?

RQ3: How do software practitioners reason about the relationship between affective states and technical debt?

The sections of this paper are laid out as follows. The following section presents related work in the research areas of TD and human aspects of software engineering, individually and jointly. Section 3 describes the research design with a particular focus on reproducibility. Next, Section 4 and Section 5 present the quantitative and qualitative analyses, respectively. The study is discussed in Section 6, limitations and threats to validity are presented in Section 7, and the paper is concluded in Section 8.

2 Related Work

Much of the current literature on Technical Debt (TD) pays particular attention to its technical or financial perspectives. Breaking with such traditions, this study observes TD through the lens of human aspects of software engineering. Hence, for full appreciation, the reader should be familiar with the background of two research branches.

Recounted firstly is previous research on TD in general. Appropriate nomenclature and central findings are outlined before introducing the specific type of TD investigated in this study. Next, we describe software engineering research on human behavior, emphasizing recent studies on the topic of feelings, emotions, and moods, and the recommendations concerning measurement instruments from psychology. One of those instruments, the Self-Assessment Manikin (SAM), was employed in this study and is explained in detail.

Once these two branches (i.e., the research area to be broadened, and the facet used to do so) have been covered, related works are listed. That is, existing research items that have used similar lenses and investigated challenges encountered in the TD literature. Those items are briefly reviewed to clarify how this study fits into existing research.

2.1 Previous Research on Technical Debt

Technical Debt (TD) was first conceptualized a few decades ago by Cunningham (1992) as a financial metaphor for how early misunderstandings of a problem

domain might hamper future development, lest the software is refactored to incorporate knowledge gained (Cunningham, 2009). Since then, the term has received much attention in both academia and industry. Today, the metaphor is widely used as a communicative aid for explaining (internal) software quality problems to non-technical stakeholders by emphasizing to what extent the software must compromise its ability to meet the needs of the future in order to meet the needs of the present (Cunningham, 1992; Avgeriou et al., 2016; Ampatzoglou et al., 2015; Fernández-Sánchez et al., 2017; Ernst et al., 2015).

One of the main strengths concerning the definition of TD is that much of its terminology originates from finance. As noted by Ampatzoglou et al. (2015), the two most commonly used terms in TD research are *principal* and *interest*, i.e., the cornerstones of financial debt. In software engineering, the former expresses the effort required to turn the current quality of some development artifact into its optimal level—the latter concerns how this sub-optimal level of quality leads to extra effort in later development iterations.

Several studies have shown that TD has significant negative consequences that can be detrimental to software companies (Tom et al., 2013; Li et al., 2015; Besker et al., 2018a; Ampatzoglou et al., 2015; Fernández-Sánchez et al., 2017). The interest does away with a substantial portion of development time (Besker et al., 2017, 2019), and may grow non-linearly if left unattended (Martini and Bosch, 2017). Further, TD tracking and TD management are uncommon in the software industry, and when encountered, the processes are typically immature (Guo et al., 2011; Ernst et al., 2015; Martini et al., 2018a).

Despite its severity, TD is difficult or impossible to measure directly, and assessments typically rely on measurement proxies known as software smells, i.e., indicators of (internal) software quality issues (Fontana et al., 2017; Ganesh et al., 2013; Garcia et al., 2009; Sharma and Spinellis, 2018). Naturally, empirical studies, such as this one, face the same issue when they need to exemplify TD items.

So far, we have outlined the previous research on TD in general, by giving an account of its history, terminology, and critical findings. The next paragraphs will focus on a type of TD known as design TD (DTD), as it is the category that our investigation is based on.

True to its name, DTD is TD found in software design, i.e., sub-optimal constructs in the software system's structure and behavior. As such, its boundary to, e.g., architectural TD, is disputed. Some researchers merge the two (Tom et al., 2013). Others separate them (Li et al., 2015; Alves et al., 2016) according to definitions that typically are too vague or subjective to form disjunct sets (Alves et al., 2014, 2016).

Such disagreements propagate to the categorization of software smells (Garcia et al., 2009), which results in some smells, e.g., cyclic dependencies and hub-like dependencies to be considered either design smells (Ganesh et al., 2013) or architectural smells (Fontana et al., 2017).

Although this study does not intend to nuance said discussions, it is still important to clarify how those terms are used here. Our stance is that diffuse boundaries are still preferable to a lack of distinction in the absence of clear definitions. Many recent studies have named ATD as particularly important (Ernst et al., 2015; Besker et al., 2018a), and confounding the category could potentially dilute those findings. We see DTD and design smells as small, local instances in isolated parts of the software system and can be comprehended easily.

2.2 Previous Research on Human Aspects of Software Engineering

A growing body of literature recognizes the importance of interdisciplinary research between software engineering and psychology (Cruz et al., 2015). Both academia and industry acknowledge that software engineering tasks are human activities and, thus, impacted by human aspects (Boehm and Papaccio, 1988; Feldt et al., 2010; Colomo-Palacios et al., 2010; Tamburri et al., 2013; Fagerholm et al., 2015).

For many years, such studies were dispersed, but in 2015 Behavioral Software Engineering (BSE) was proposed as a common platform for research concerned with "the study of cognitive, behavioral, and social aspects of software engineering performed by individuals, groups, or organizations" (Lenberg et al., 2015).

Out of the many tracks in this research area, one concerns *affective states* (or *affects*, for short), i.e., feelings, emotions, and moods. Previous studies have linked affects to, e.g., debugging performance (Khan et al., 2011), analytical ability (Graziotin et al., 2014), and productivity (Graziotin et al., 2015a).

This study is placed firmly within the said track and is part of a sub-field called Psychoempirical Software Engineering (PSE), i.e., software engineering studies that use theory and measurements from psychology (Graziotin et al., 2015c). This article follows the Graziotin et al. (2015c) guidelines for conducting PSE research, which also synthesizes the psychology theory concerned with affective states.

According to said guidelines, this study's objective is best met by subscribing to the *dimensional framework* and employing the *Self-Assessment Manikin* (SAM) instrument for measuring affective states (Graziotin et al., 2015c). Within the dimensional framework, affects are expressed through several distinctive dimensions, e.g., the models represent affective states along three continua: pleasure–displeasure (valence), arousal–nonarousal (arousal), and dominance–submissiveness (dominance) (Graziotin et al., 2015c; Russell and Mehrabian, 1977).

The recommended instrument, the SAM, measures affects through pictorial representations (Figure 1) of the three dimensions of the models (Graziotin et al., 2015c; Lang, 1980; Bradley and Lang, 1994; Morris et al., 2002). Developed by Lang (1980), the instrument has, over the decades, been subjected to extensive validation research (Morris, 1995) and seen used in numerous studies, see, e.g., (Morris, 1995; Betella and Verschure, 2016) for many examples.

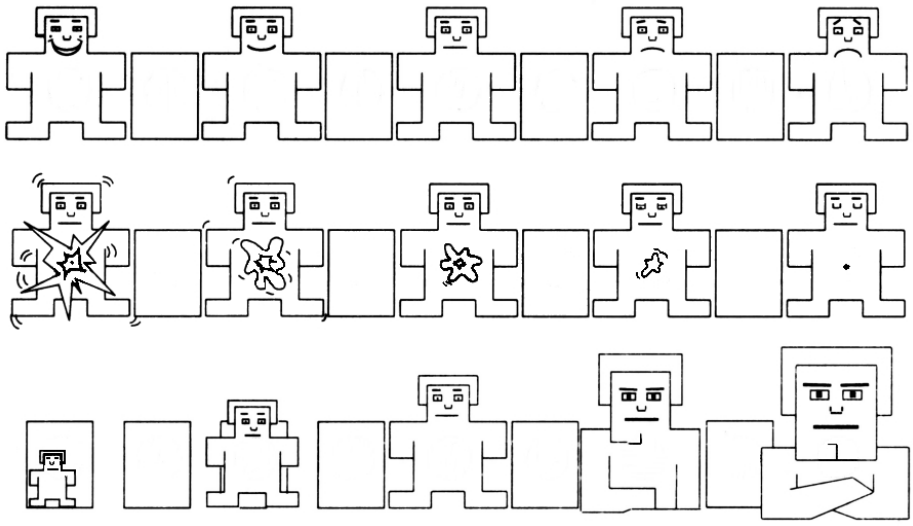


Fig. 1 The SAM measurement instrument. SELF ASSESSMENT MANIKIN © Peter J. Lang 1994

According to Bradley and Lang (1994), the graphic design of the SAM has many benefits. The lack of verbal components means that the SAM can be administered to a broader population range, including individuals with a non-English mother tongue or language disorders, and children. Additionally, the SAM can measure direct affective reactions, as it can be filled out in a short amount of time and eliminates cognitive processing (Morris et al., 2002). Further, Morris (1995) argues that the use of stylized characters, as opposed to photographs of humans, makes the SAM less susceptible to many types of biases and should, thus, be considered culture-independent.

However, because SAM relies on self-reporting, the scores are not standardized according to objective reference points. Although individuals are consistent with themselves (within measurement), the ratings cannot be assumed to be consistent between individuals (between measurement) (Graziotin et al., 2015c). In other words, two individuals could rate the same affective state in two different ways. Consequently, investigations administrating the SAM should follow a within-subject (or repeated measures) design (Graziotin et al., 2015c).

The SAM is protected by copyright law, but the instrument and instructions for proper administration (Lang et al., 1997) is available for use for academic, not-for-profit research[1].

[1] Information about how to obtain the SAM can be found at `https://csea.phhp.ufl.edu/Media.html`

2.3 Interdisciplinary Research on TD and Human Aspects

Data from several secondary studies reveal that few TD studies have investigated the relationship between TD and human aspects (Tom et al., 2013; Li et al., 2015; Ampatzoglou et al., 2015; Alves et al., 2016; Fernández-Sánchez et al., 2017; Besker et al., 2018a). Instead, the technical and financial aspects are what is investigated, e.g., software quality or cost of future changes.

When human aspects are addressed in TD research, morale is the most frequent question. A negative correlation was early proposed by Tom et al. (2013), based on anecdotal evidence found in web blogs. Since then, empirical investigations have corroborated the connection, including previous articles of our own.

Spínola et al. (2013) performed a survey on TD folklore and found medium to high consensus among software practitioners that TD is related to their morale. In conjunction with interviews, a survey was also carried out by Besker et al. (2020) to determine what the effects of occurrence and management of TD are on developers' morale. Their findings show that the existence of TD negatively impacts morale, but also that morale is increased by proper TD management.

Although a common misconception, morale is not the same thing as affective state (Graziotin et al., 2015b). There are no previous TD studies that investigate affects and even fewer that directly measure how software practitioners respond to TD items to the best of our knowledge.

In addition to morale, some empirical studies have offered evidence for TD harming the software practitioner's psychology. Lim et al. (2012) found that developers are more reluctant to incur TD because its consequences become a part of their daily work. Similarly, such reluctance may arise due to developers predicting the sub-optimal construct needs to be corrected sooner or later, and that task would fall on them (Yli-Huumo et al., 2014). However, these findings were somewhat opportunistic and limited, as neither study set out with the research objective of investigating such questions.

As can be seen, TD research has thus far shown lukewarm interest in the relationship between TD and human aspects. However, the topic has also been approached from the PSE direction, and those studies present interesting empirical findings. Graziotin et al. (2017) surveyed software practitioners concerning causes for unhappiness, and established that low code quality and coding practices, and being stuck in problem-solving, were among the most significant factors. Additionally, Graziotin et al. (2018) investigated the adverse effects of developer displeasure and found, among many other types of consequences, *lower code quality* and *discharging code*.

Not only are these factors intimately connected with TD, but they pose the threat of vicious cycles: low code quality causes unhappy developers, and unhappy developers produce low quality code. Unfortunately, the studies did not drill down into this problem, which could answer questions such as its probability and severity. Nor was the issue approached specifically from the TD perspective. Clearly,

our study differs from the previous PSE studies, as it seeks to investigate affects regarding specific TD items.

In conclusion, prior research shows that investigating human aspects concerning TD is a promising prospect. To effectively manage TD, we need to understand better how software practitioners, as human beings, can be factored into the trade-offs between short-term and long-term benefits. Despite this, the current body of knowledge is limited, and both academia and the software industry would likely benefit from further clarification.

3 Methodology

As suggested in the previous section, our research topic has received little attention despite interesting initial findings. Consequently, the study design must acknowledge the limitations posed by such research gaps, e.g., validation against previous findings may be impossible.

One of the countermeasures implemented in our design is choosing a mixed-methods approach, i.e., collecting both quantitative and qualitative data. This decision is appropriate because it enables the study to improve validity, e.g., the results from one analysis could corroborate or rebut findings from the other. In this study, data were gathered from three sources: a repeated-measures experiment (quantitative), a questionnaire (quantitative), and a semi-structured interview (qualitative).

Another central countermeasure is transparency. While this section strives to describe the planning and implementation with enough clarity to make validity threats evident, we will actively use this space to highlight identified validity threats and explain how those were managed. Further, a replication package is made available.[2] It contains complementary information and all material needed for reproducing the study, as it is infeasible to present all details within the scope of this article.

3.1 Goals

This study seeks to examine the relationship of design smells on software practitioners' affective states. As such, it tries to understand the importance of human aspects as a factor in TD. Among other things, we hope that the answers to our research questions will spark further interest in considering software practitioners when making trade-offs between short-term and long-term benefits. This goal begs for persuasive evidence, which can be provided through empirical research.

[2] `https://github.com/torkar/affective_states`

RQ1 (see Section 1) will be answered through experiments and the principled use of Bayesian statistics, where we employ Markov chain Monte Carlo to sample multi-level models with a within design of subjects. This research question aims to investigate the actual relationship between affects and DTD, without being colored by the participants' (nor the researchers') preconceived notions. As for delimitations, this RQ will examine a handful of design smells and consider affects from the presented models' perspective alone.

The motivation behind RQ2 is to see what role individual differences play. Because the study examines affects, the experimental units must be human participants, which opens up many exciting characteristics that could be studied. However, while data for various factors could be collected with ease, there are trade-offs to consider, e.g., transparency and confidentiality. Since the data are open (see the replication package), many characteristics that could easily identify an individual (e.g., gender or ethnicity) were not recorded.

Finally, RQ3 was included to understand the topic's appearance in the software industry. Hence, this research question is broader than the other two and of a more exploratory nature. Giving voice to the practitioners' reflections on affects and TD can increase understanding in a broader context and reveal peripheral issues.

3.2 Study Design

As this study collected three sets of data, its design is a substantial part of this article. Since there are many constructs to keep track of and to make them clear, we will use a few different viewpoints. The first viewpoint is that of *sessions* and is modeled in Figure 2.

From this perspective, the study was designed as 90-minute sessions, one for each participant. At the start of their session, the participant received instructions (pre-task instructions) outlining the study and its session. The participant obtained these in three steps:

1) reading, understanding, and signing a document describing the treatment of, and their rights regarding, collected data (confidentiality assurance);
2) listening to instructions for, and seeing examples of, how to use the measurement instrument—which relies on self-reporting (SAM instructions) and;
3) hearing a description of what activities they will perform during the experiment (task description).

Next, during the second part of the session (measurement sitting), quantitative data were collected from a repeated-measures experiment. For this part, as well, the participant went through three steps (please note that being of a repeated-measures design, the second and third steps were conducted five times):

1) using the measurement instrument on a practice task (anchor point);

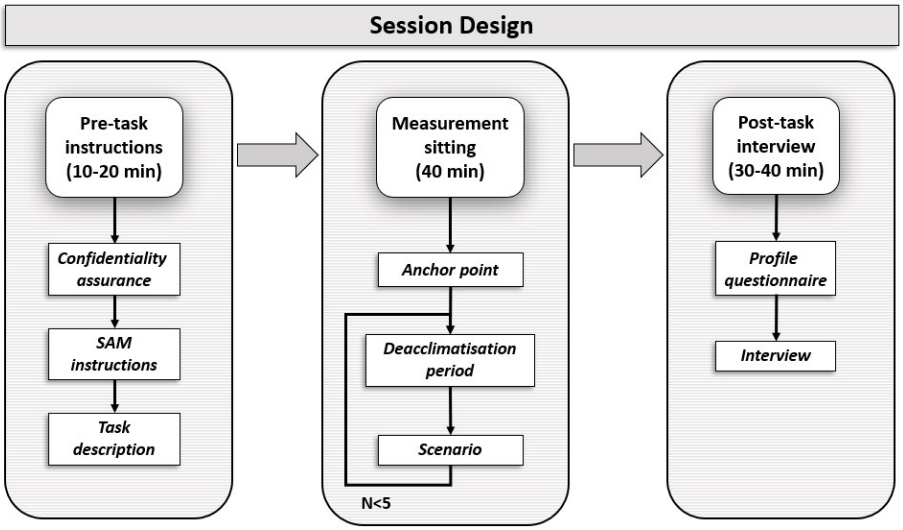


Fig. 2 The session view of the study: 90-minutes sessions, conceptually comprising three parts with eight steps.

2) pausing briefly (deacclimatization period) and;
3) using the measurement instrument on a task (scenario).

In the last part (post-task interview), the two remaining data sets were gathered: quantitative data from a questionnaire and qualitative data from a semi-structured interview. In a straight-forward manner, these were presented to the participant in one step each:

1) filling out answers to questions about their professional experience with software (profile questionnaire) and;
2) talking and answering questions about how they perceived the study and their view of code maintainability and feelings (interview).

Thus, the session perspective is concluded. This description has given an overview of what the participants did, between being greeted by the researchers to saying goodbye. It also introduced concepts that are key to understanding the study design but did so on a high abstraction level. Later in this section, the concepts will be revisited for additional detail.

Next, we consider the study from the perspective of *data collection*. Three sources of empirical data (experiment, questionnaire, and interview) were gathered from the participants. As shown in Figure 3, each of these data corpora were designed around one of the RQs, i.e., the experiment for RQ1, the questionnaire for RQ2, and the interview for RQ3. Similarly, the experiment data and the questionnaire were modeled in the same statistical analysis, while the interview data underwent thematic analysis.

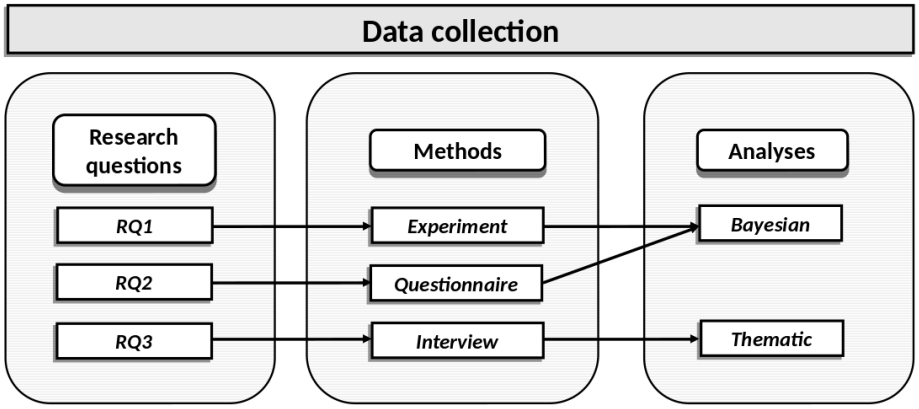


Fig. 3 The relationships between the RQs, methods, and analyses.

ID	Smell	Smell category
ScA	Missing Encapsulation	Encapsulation smell
ScB	Missing Hierarchy	Hierarchy smell
ScC	Broken Modularization	Modularization smell
ScD	Cyclically-Dependent Modularization	Modularization smell
ScE	Rebellious Hierarchy	Hierarchy smell

Table 1 The scenarios used in the experiment and the smells they embody.

First, the experiment set out to understand the relationship between affects and DTD. From this goal followed that, ideally, all factors except for the amount of design debt (explanatory variable), should remain constant. Then, what was measured was the participants' affective state in terms of valence, arousal, and dominance (response variables).

However, since the experiment was of the repeated-measures variety, its design was more complicated. While the explanatory variable still represented the amount of design debt, there was not one, but five such variables (one for each repetition, or *scenario*). In other words, as the participant progressed through the experiment, they would encounter five different scenarios: ScA, ScB, ScC, ScD, and ScE. Within each scenario, the participant received one treatment and then reported their affective state.

Because design debt is difficult to measure, each response variable had two levels and represented whether its design smell (see Table 1) was present or had been refactored away. That is, the scenario variant where the smell had been removed had a lower (L) amount of technical debt than its partner variant (H).

Moving on to the second method, the questionnaire aimed to investigate how professional characteristics factor into the participants' responses. The questions are listed in Table 2.

ID	Type	Description
Q1	Closed	My highest level of completed academic education is _____
Q2	Closed	My education major (e.g., computer science, electrical engineering, software engineering, ...) was _____
Q3	Closed	I have working experience with software for _____ years.
Q4	Closed	My current role (e.g., architect, developer, tester, ...) is _____
Q5	Closed	The programming language I am most experienced in is _____
Q6	Closed	My currently preferred programming language is _____
Q7	Closed	Most of my working experience comes from the following domain (e.g., telecom, healthcare, finance, ...) _____
Q8	Open	Do you have any additional comments concerning this questionnaire?

Table 2 The questionnaire questions.

ID	Type	Description
IQ1.1	Open	Could you please tell us more about your daily work. What type of tasks do you normally encounter?
IQ1.2	Open	How do those tasks make you feel?
IQ1.3	Closed	Do you face challenges in those tasks?
IQ1.4	Open	How do those challenges make you feel?
IQ1.5	Closed	Are those feelings frequent?
IQ2	Open	In contrast to challenging tasks, what sorts of feelings would you say you get from routine tasks?
IQ3	Closed	Do you think that anything outside of this experiment did impact your responses today?
IQ4.1	Open	Would you please tell us how you experienced the code examples?
IQ4.2	Open	What about the software design in the examples?
IQ5	Open	What would you say are the differences between the scenarios we provided and software one encounters in industry?
IQ6	Closed	Did you find SAM difficult to use or understand?
IQ7	Open	That was all of the questions that we had for you. Is there anything you would like to add?

Table 3 The common questions of the semi-structured interview.

The third method, the interview, was designed to answer RQ3, but also to explore the topic of TD and human aspects beyond the delimitation of this study. That is, it would have been a waste of resources to constrain the participants to talk merely about DTD when there are so many other facets left to investigate. Further, bounding the interview responses too strictly could limit the participants' divergent thinking and, thus, the data's richness. Naturally, this risk is more severe because of the limited amount of previous research, which could otherwise have been drawn upon to guide the interviews.

Instead, the participant was allowed to speak more or less freely about their perception of affects and software maintainability. The questions listed in Table 3 were asked at opportune times during the interview, to light-handedly steer it. These were complemented by probing questions, i.e., follow-up questions to the participant's reasoning.

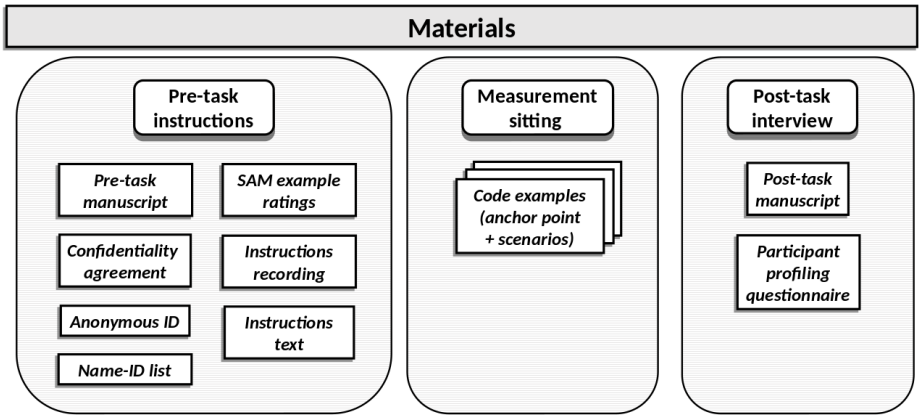


Fig. 4 The experimental materials used in the different parts of the session.

As mentioned earlier, the interviews had a broader scope than this study. Hence, the thematic analysis used to answer RQ3 is considered a subset (highlighted) of the interview questions, namely IQ4.1, IQ4.2, and IQ5.

Thus, the data perspective is concluded. It presented how the research questions can be traced to the selected methods and analyses. Further, the general structure of the methods was explained, including the questions asked to the participants.

The third and final perspective is the *materials* perspective, which is illustrated in Figure 4. Their description is deferred to the replication package, where also the experiment protocol is included.

3.3 Participants

Forty software practitioners participated in this study. They attended one 90-minute session each and did so between March 26 to April 17 (2019). The session was held in a conference room at their work office (four cities in Sweden).

The participants were employed by eleven companies and one government agency, which covered a diverse set of company sizes and business domains (e.g., automotive, finance, and renewable energy).

The participants were selected by the companies, which in turn had been selected through convenience sampling by using the researchers' and the university's professional networks. The companies were then asked to select participants at their own volition, as one of several actions to reduce the risk of discriminating roles or company types. Consequently, the companies contributed with as many participants as they wanted (range 1–7, $\mu = 3.33$, $\sigma = 1.87$). Similarly, the term

software practitioner (as opposed to, e.g., *developer*) was used to describe eligible participants.

3.3.1 Participant Motivation

Voluntary participation was motivated by informed consent and anonymity, i.e., neither participants nor companies were offered monetary or similar benefits. In return, we made sure to limit any economic harm, e.g., by terminating sessions that would continue past the allocated time. Hence, participant commitment was comparatively minor: a maximum of 90 minutes of their regular work time, approved by their company. Further, the participants were informed about their rights to decline to answer any questions or, at any time, withdraw from the study.

Informed consent was achieved by the participants reading and signing a confidentiality agreement (available in the replication package) at the start of the session. It disclosed the study's purpose, promised anonymity, and declared that the participant's data would be destroyed upon their request.

Confidentiality was ensured through randomly generated identifiers. The identifier was linked to the participant through a single (physical) document, which would be used if required during the analyses, e.g., to explain potential outliers. The links were destroyed upon completion of the analysis phase.

In conclusion, the participants can be presumed to have taken part in this study willingly. They faced no coercion, and their employer authorized participation.

3.3.2 Allocation to Treatments

The repeated-measures experiment consisted of five scenarios (ScA–ScE) with two levels each (L and H). A genuinely random assignment (scenario permutations and random level) would result in an infeasible number of combinations. Hence, the levels were fixed to two *treatment patterns*: LHHLH and HLLHL. In other words, the n^{th} participant was randomly allocated to a permutation of the scenarios but would receive the treatment pattern LHHLH if n was odd and HLLHL if n was even.

Those particular (complimentary) treatment patterns were constructed to mitigate the risk of them influencing responses. For instance, some patterns would obfuscate whether the explanatory variable caused response variable changes, e.g., LLHHH might confound the participant becoming bored or comfortable with the experiment. Similarly, some patterns could potentially allow participants to influence the data, e.g., the participant might recognize that LHLHL alternates the levels.

3.4 Setting

Each session included a singular participant. The researchers arrived at the room 30 minutes before the session to ensure ample time for preparations. Further, this time would act as a buffer if multiple participants from the same company were scheduled in sequence and, e.g., arrived late.

3.5 Anchor Point

At the start of the measurement sitting, the participant was presented with a practice scenario (anchor point). The intention was to accustom the participant to the experiment, thereby lowering the impact of learning effects.

From the viewpoint of the participant, the anchor point should appear no different from the scenarios. Their SAM ratings for the anchor point were, however, not included in the analysis. Further, to act as a common baseline for all participants, the anchor point had just one treatment level (H).

3.6 Deacclimatization Periods

Evidence suggests that affective states may persist after stimulus removal (Gomez et al., 2009). Hence, the repeated-measures could be distorted if the participant carried over affects from one scenario to the next. To help the participant return to their natural affective state, short pauses (deacclimatization periods) were introduced between each scenario.

Surprisingly, a search of the literature revealed few studies discussing the topic of deacclimatization periods—none of those offered or referenced concrete guidelines, e.g., duration or appropriate activities. For instance, Cinaz et al. (2013) used relaxing documentary films but did insufficiently report on the procedure and the material to allow reproduction.

In the end, we employed 120 seconds of rest and no specific activity (silence or small talk).

3.7 Scenarios

As mentioned previously, DTD is challenging to measure directly. Hence, software scenarios were constructed to act as proxies during the experiment. Special care was taken to create appropriate scenarios, here called *concrete representations* of DTD. First, to make the findings more relevant to industry, the scenarios should

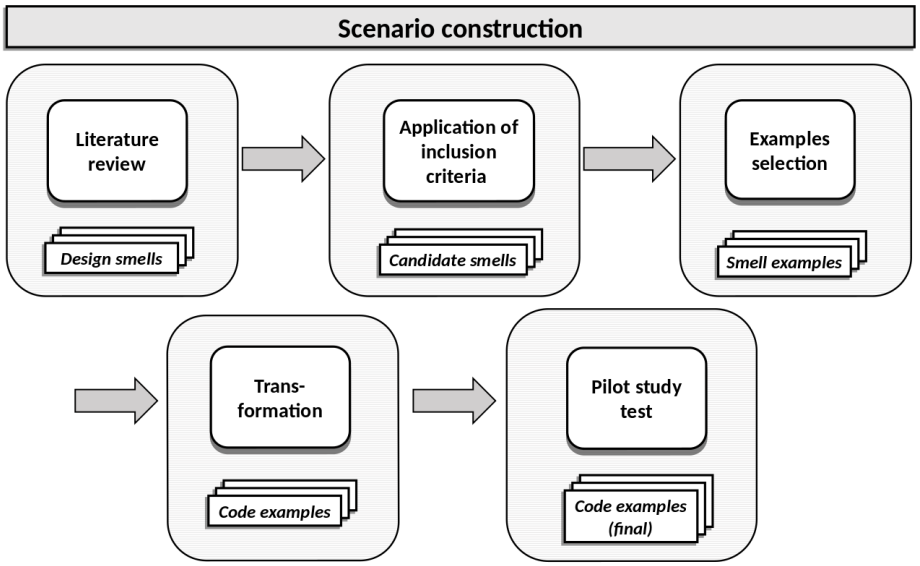


Fig. 5 The process for constructing the scenarios.

exemplify a problem that could realistically be encountered in practice. Second, because smells are not necessarily indicative of definite quality problems (Sharma and Spinellis, 2018), the design smells should be actual (rather than contingent) liabilities. Third, because the proxies should be self-contained, it must be possible to present the TD item (and its resolution) in a reasonably sized piece of source code.

In other words, the advantages of the refactoring of these *concrete representations* of DTD should be pretty much evident from the examples alone (keep in mind, however, that the participants cannot compare the two versions). Unfortunately, concrete representations differ from the abstract representation of smells typically used in smell catalogs, see, e.g., (Garcia et al., 2009).

After reviewing the literature, the scenarios were constructed (see Figure 5) based on the work of Ganesh et al. (2013), and refined by Suryanarayana et al. (2014). Out of the 25 design smells, we selected (candidate smells) those that fulfilled the following inclusion criteria:

1) The smell had a *concrete representation* and;
2) The smell was listed as negatively impacting understandability and;
3) The essence of the smell could be understood in a short amount of time.

In instances where Suryanarayana et al. (2014) listed several examples (smell examples) of the candidate smell, we chose the one deemed most appropriate for the experiment's context.

Next, the smell examples were transformed into code examples. They were harmonized into correct syntactic syntax highlighted, Java source code that conformed to a popular coding style guide. Some examples were deemed too domain-specific without explanatory texts. Those had their context and names modified.

Finally, these code examples were tested in pilot studies, and the six (anchor point plus five measurements) most suitable scenarios were used for the measurement sitting.

3.8 Analysis Procedure

Being a mixed-methods study, two different analyses were performed. For the quantitative part, a Bayesian statistical model was implemented and executed in *R* (R Core Team, 2019). The procedure is fully available in the replication package.

The qualitative data was analyzed by following the guidelines for thematic analysis by Braun and Clarke (2006). Thematic analysis is frequently applied in both psychology (Braun and Clarke, 2006) and software engineering (Cruzes and Dybå, 2011). It is a flexible approach tailored to the research needs.

The flexibility of thematic analyses stems from several choices that the researchers must make when deciding how to conduct the analysis (for a discussion about each choice's advantages and disadvantages, please see (Braun and Clarke, 2006)). For this study, the analysis was *inductive*, searched for *semantic themes* and theorized *essentialistically*. In other words, we coded the interview transcripts in a data-driven fashion without trying to fit into a pre-existing coding frame. Themes were then identified and interpreted based on what was explicitly articulated within the data set.

The primary reason for these decisions is the small amount of previous research on the relationship between TD and human aspects of software engineering. For example, the *inductive* approach does not rely on existing theory to the same extent as the *theoretical*. Similarly, it seemed more prudent to identify the themes at the *semantic* level, given the exploratory nature of this investigation. Otherwise, the likelihood of projecting personal beliefs onto *latent* themes could be excessive. The same reasoning underpinned the choice of performing an *essentialist* analysis. In particular, previous research on human aspects of TD did not seem to lend sufficient support for theorizing socio-cultural contexts and structural conditions (beyond little more than pure speculation), as is sought with the *constructionist* perspective.

Since the qualitative analysis aimed to discover the most central ideas and themes (rather than most, or all of them), the analysis' size was determined by salience rather than saturation. This decision is somewhat uncommon in software engineering research, so a short motivation is in order.

Salience is the idea of analyzing qualitative data regarding the most prominent items, and can be contrasted with saturation, i.e., until the set of all unique items is *believed* to have been exhausted. For a broad range of research objectives, saturation would be superfluous, as salient items are, unsurprisingly, more prevalent and more culturally significant than non-salient items (Weller et al., 2018). In other words, many research questions can be answered with smaller sample sizes than what would be required to claim saturation.

Because 10 interviews are sufficient to reliably capture up to 95 % of the most salient ideas (Weller et al., 2018), that number of data items were randomly selected for the data set (out of the 39 items in the data corpus).[3] Indeed, this study's necessary sample size might be even lower, as we used probing techniques during the interviews, e.g., repeating phrases the interviewee uttered when working with the scenarios and asking for more information.

4 Quantitative Analysis and Results

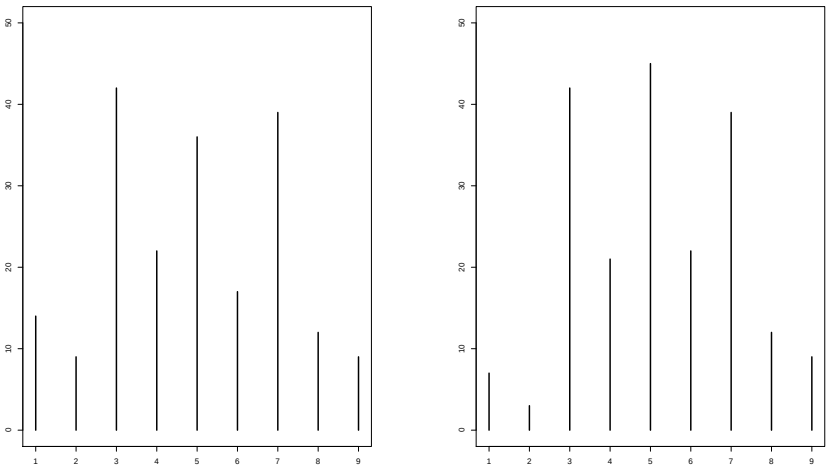
Forty subjects participated in the experiment, and each subject contributed with five measurements to estimate our outcomes. In addition, the following data were collected: educational level (e.g., bachelor), the example used (the ten experimental artifacts, i.e., five artifacts in L and H setting), academic major (e.g., computer science), role (e.g., designer), language experience (e.g., Java), entities (i.e., level of complexity of the artifact), and years of work experience. The latter was scaled in order to improve sampling.

Given the three outcomes valence, arousal, and dominance $\{V, A, D\}$, and the predictors listed above, the data consists of a matrix with 200 observations (rows) and 11 variables (columns), with no missing values (Figs. 6a–6c provide an overview of the outcomes).[4]

In this analysis, we employed Bayesian ordinal regression, using a cumulative model (for an introduction to Bayesian analysis, please see (Furia et al., 2019)). One could imagine two other potential models, i.e., the sequential model or the adjacent category model. However, since Likert (1–9) scales were used, cumulative models are more suitable. i.e., the sequential model would be suitable if we want to analyze the number of correct designs predicted from experience. In contrast, the adjacent category model would be appropriate if we want to predict the number of correctly solved sub-items of a complex task—none of this was of interest to us (Bürkner and Vuorre, 2019). Several models were designed, and their relative out of sample prediction capabilities was evaluated iteratively. The final model, below, includes all relevant predictors and has the same out of sample capabilities as other comparable models. For model comparison, we used state of the art model

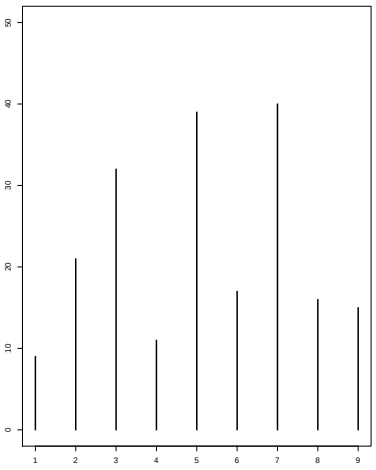
[3] A singular participant asked not to be recorded during the interview and could thus not be included.

[4] The dataset, with analysis scripts and a `Docker` image, can be found at `https://github.com/torkar/affective_states`. `R` 4.0.2, `rstan` 2.21.2, and `brms` 2.13.9 was used for the analysis (R Core Team, 2020; Bürkner, 2017, 2018; Stan Development Team, 2020)



(a) Valence (V)

(b) Arousal (A)



(c) Dominance (D)

Fig. 6 Histograms of the outcomes $\{V, A, D\}$. On the x-axis, we have the responses (Likert 1–9), and on the y-axis, we have the frequency.

evaluation (Vehtari et al., 2017).[5] Next, follows the design of the final model and the corresponding priors.

$$
\begin{aligned}
V_i, A_i, D_i &\sim \text{Cumulative}(\phi_i, \kappa) && (1)\\
\phi_i &\sim \beta_1 \text{EDUCATION}_i + \beta_2 \text{EXAMPLE}_i + \beta_3 \text{MAJOR}_i + \beta_4 \text{ROLE}_i && (2)\\
&\quad + \beta_5 \text{LANGUAGE}_i + \beta_6 \text{ENTITIES}_i + \beta_7 \text{EXPERIENCE}_i && (3)\\
&\quad + \beta_{\text{SUBJECT}[i]} && (4)\\
\beta_1 &\sim \text{Dirichlet}(2, 2, 2, 2, 2) && (5)\\
\beta_{\text{SUBJECT}} &\sim \text{Half-Cauchy}(0, 2) && (6)\\
\beta_2, \ldots, \beta_7 &\sim \text{Normal}(0, 0.5) && (7)\\
\kappa &\sim \text{Normal}(0, 5) && (8)
\end{aligned}
$$

As is evident from the first line, we model each outcome, $\{V, A, D\}$, using a cumulative likelihood. The parameters ϕ and κ are the linear regression and the intercepts, respectively, which we model for each outcome (i.e., we have eight intercepts for each outcome since the outcome was Likert scale 1–9).

In the next three lines, we have the linear regression. We have eight parameters we want to estimate, one for each of our predictors. The parameters β_1 and $\beta_{\text{SUBJECT}[i]}$ are special as we will see next.

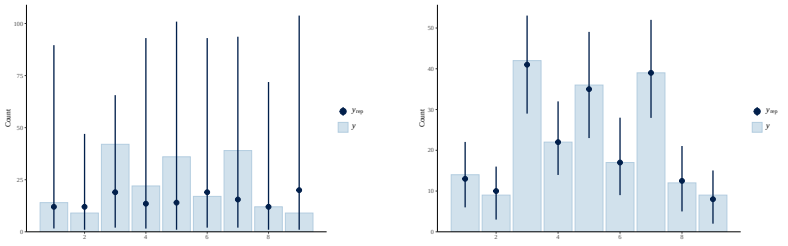
On Line 5, we assign β_1 a Dirichlet prior. The Dirichlet prior is the multivariate generalization of the Beta distribution (a distribution commonly used to model a probability $[0, 1]$). Using Dirichlet, we can model an array of probabilities, i.e., in this case, we model five probabilities and use a very weak prior (the 2s), indicating that we do not have any prior knowledge. The reason we use a Dirichlet here is monotonicity, i.e., the predictor EDUCATION is an ordered categorical variable indicating level of education. We, thus, want to model the probability separately for each of the categories in education.

Continuing on Line 6 we assign β_{SUBJECT} a Half-Cauchy$(0, 2)$ prior. This prior is common when modeling standard deviations and allows only positive real numbers ($\mathbb{R}^+$). To analyze variability in this way goes by many names, e.g., random effects or varying intercepts. The reason for why we use it is due to us following the latest recommendations by designing the experiment to collect within-person measurements (Leek et al., 2017), i.e., each subject has been randomly allocated several tasks and, thus, we model the variability of each subject to partially pool the estimates, to avoid overfitting.

Proceeding to Line 7, we assign the priors Normal$(0, 0.5)$ for the remaining parameters while, on the last line, we assign the prior Normal$(0, 5)$ to all intercepts for each outcome. (It is common to assign a broader prior for intercepts.)

The careful reader would react to what could be perceived as tight priors for several parameters, i.e., Normal$(0, 0.5)$. However, first, using Normal$(0, 0.5)$ on six

[5] Pareto $k < 0.5$ and LOOIC $= 2406.0$.



(a) Prior predictive check (b) Posterior predictive check

Fig. 7 Prior and posterior predictive checks (y is the data, and y_{rep} are 100 draws from the probability distribution). The left plot shows the prior predictive checks. As is evident, the uncertainty is considerable, and the median values are approximately the same for all items on the Likert scale. Compare this to the right plot, where we have drawn samples from the posterior probability distribution, i.e., we have sampled our model with data, and the data has swamped the prior probability distribution.

	Est.	Est. Error	l-95% CI	u-95% CI
σ_V	0.39	0.24	0.02	0.90
σ_A	0.87	0.27	0.32	1.42
σ_D	0.60	0.26	0.07	1.11

Table 4 Standard deviations of random effects.

parameters still makes an impressive standard deviation, $(6{*}0.5)^2 = 9$, and, second, the combination of all priors established *a nearly uniform prior* on the probability scale, i.e., prior predictive checks, and a sensitivity analysis was conducted.

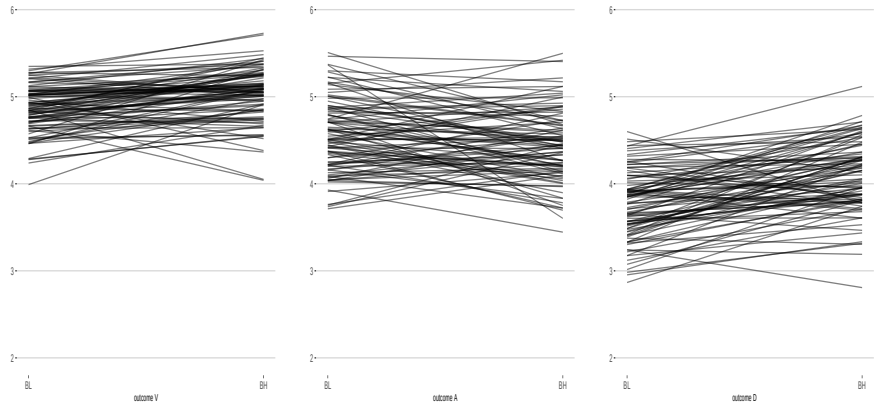
Since we used Markov chain Monte Carlo (more specifically Hamiltonian Monte Carlo) to sample, we also had several diagnostics. In our case, the model showed no indications of a biased posterior, and diagnostics ($\widehat{R}$, effective sample size, and trace plots) indicated that the chains had converged. Posterior predictive checks clearly showed that the data swamped the priors (see Figs. 7a–7b for a visualization of the prior predictive checks and posterior predictive checks).

Continuing this section, we will next look at the output from the model. First, we will present the standard deviations for each outcome's random effects and any interesting population-level effects. Then, we will predict outcomes while fixating specific parameters. The final part will present the results of the hypothesis testing (Bayes factor).

Analyzing the random effects (the varying intercepts), we see that there is not much difference in the uncertainty of the estimates concerning σ for our three outcomes, as the standard deviations' credible interval mass vary from 0.88 (σ_V) to 1.1 (σ_A) (Table 4). In short, the uncertainty for each outcome, $\{V, A, D\}$, is very much the same, but, notably, valence (V), has a standard deviation, $\sigma = 0.39$, while arousal (A) has the largest standard deviation, $\sigma = 0.87$.

Outcome	Parameter	Est.	Est. Error	l-95% CI	u-95% CI
Dominance (D)	EXAMPLE (BL)	−0.78	0.34	−1.43	−0.12
Valence (V)	EXAMPLE (BH)	0.73	0.34	0.07	1.39
Valence (V)	EXAMPLE (DL)	−0.83	0.35	−1.52	−0.14
Valence (V)	EXAMPLE (CL)	0.72	0.36	0.02	1.42
Valence (V)	EXPERIENCE	0.25	0.16	−0.05	0.56

Table 5 Parameters of interest.



(a) (b) (c)

Fig. 8 From left to right 100 samples drawn from our posterior probability distribution for the outcomes Valence (a), Arousal (b), and Dominance (c). On the x-axis, we move from Low to High, while the y-axis displays the Likert scale response. This way, we simulate the outcomes given a set of fixed values. As is clear, even though these parameters were significant, if fixated to a representative sub-sample it shows that there is much uncertainty, albeit we see some positive trends.

Analyzing the estimates, and their corresponding 95% credible intervals, led to five estimates being singled out as interesting (Table 5). Four were significant on the arbitrary 95%-level, while one is positive, albeit not significant on the 95%-level.

Since Experience has much probability mass on one side of zero ($[-0.05; 0.56]$), we will analyze it further to understand its predictive ability better. First, let us analyze the four significant effects.

If we draw 100 samples from our posterior probability distribution, we will receive a better feeling for the uncertainty concerning the four significant parameters, while fixating all other parameters. In Fig. 8, we have set our parameters to median values (or the reference category in our sample), i.e.,

- Major: Software engineering ($n = 90$)
- Educational level: Bachelor ($n = 85$)
- Experience: 8 (median)

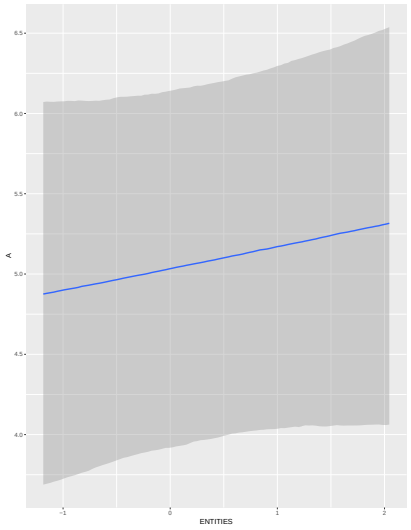


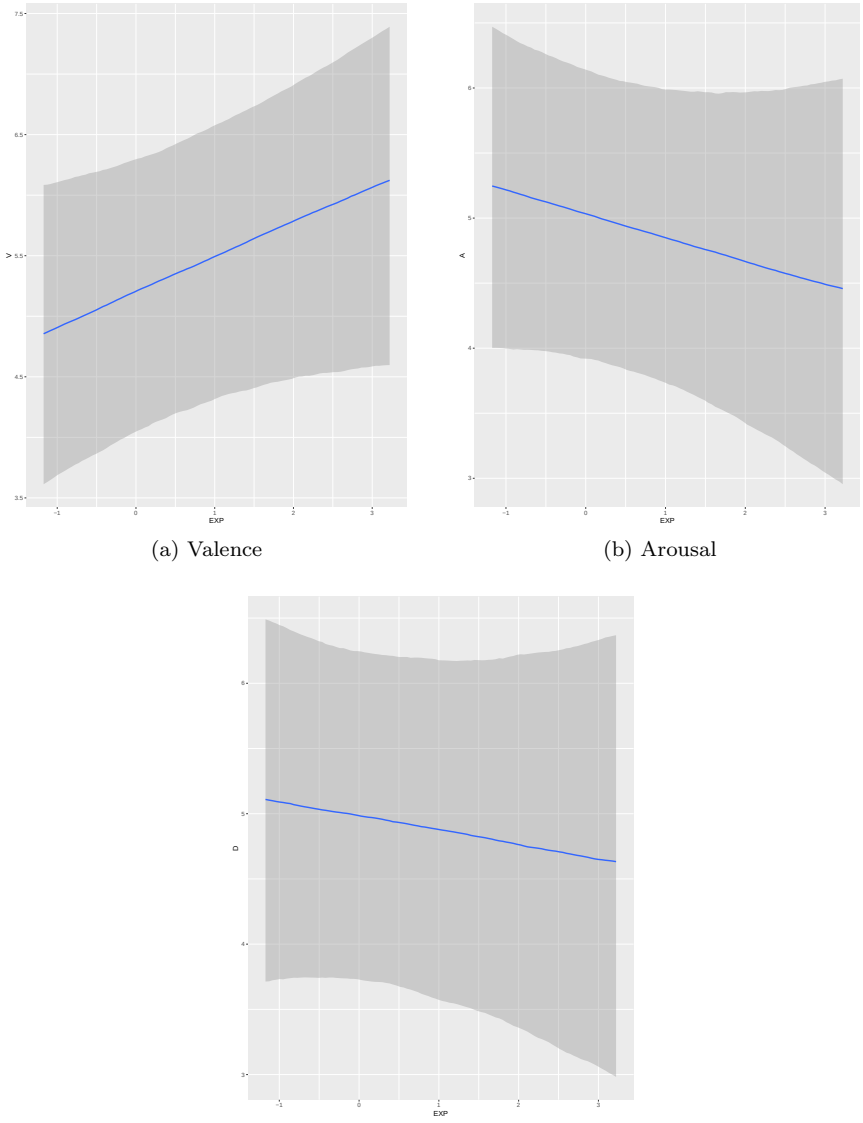
Fig. 9 Conditional effect of Entities in the model. As is evident, albeit not significant, the more complex an entity (i.e., the more to the right we move on the x-axis), the higher the outcome on the Likert scale (y-axis). In this case, we looked at the outcome A (arousal), but the same trend is visible in all three outcomes. The x-axis has been scaled, with 0 corresponding to median complexity. (The line is the median, while the gray area is the 95% credible interval.)

- Role: Developer ($n = 125$)
- Language: C# ($n = 100$)
- Number of entities (complexity): 5 (median)

Next, we should look at what role Entities (the complexity of each task) has in the model—even though it was not significant. From now on, we set all values to their mean for continuous variables, while the reference category is used for factors. As is evident in Fig. 9, we see a clear positive trend, which indicates that the model has been able to capture the role complexity plays correctly.

Finally, we would like to see the role Experience plays by analyzing it more carefully. If we turn our attention to Figs. 10a–10c, we see that the role it plays differs, depending on our outcome. For valence (V), we have a positive effect, i.e., the more experienced the subject, the higher the response on the Likert scale, while the opposite holds for Arousal (A) and Dominance (D). Here, it is crucial to keep in mind the direction of the SAM, i.e., an increase in V score means more displeasure; arousal increases as A decreases; low D scores denote submissiveness.

Having analyzed the conditional effects, we now turn our attention to measuring the strength of the evidence we have gathered. Our tests will *not* examine the significant population-level effects, which we list in Table 5; after all, we know that they are significant on the traditional 95%-level. Instead, we will focus on the contrasts between Low (L) and High (H) settings for our predictor Example. This means that we can present the results as several hypothesis tests (15 in total). Since



(a) Valence

(b) Arousal

(c) Dominance

Fig. 10 An overview of the conditional effects on Experience, given our three outcomes $\{V, A, D\}$. Lines correspond to the median, while the gray area is the 95% credible interval.

Symbol	Evidence ratio	Description
**	> 10	Strong evidence for H1
*	3–10	Moderate evidence for H1
?	1–3	Anecdotal evidence for H1
?	1/3–1	Anecdotal evidence for H0
*	1/30–1/10	Moderate evidence for H0
**	$< 1/10$	Strong evidence for H0

Table 6 Decision thresholds for hypothesis testing using Bayes factor, according to Kruschke (2010).

we have a posterior probability distribution, we do not have to, generally speaking, worry about multiple tests, which is often the case in a frequentist setting (Gelman and Tuerlinckx, 2000; Gelman et al., 2012).

For hypothesis testing, we will use Bayes factor to avoid the usage of p-values and, thus, to receive verdicts both in favor and against a given hypothesis (Goodman, 1999a,b). For our accept/reject decisions, we follow recommended practices provided by the thresholds presented in Table 6 (Kruschke, 2010).

Our hypothesis tests were unidirectional and, thus, tested that Low $<$ High, e.g.,

$$H_0 : \text{Example}_{\text{AL}} < \text{Example}_{\text{AH}},$$

which is to be interpreted as Example A Low is less than Example A High (and we analyze this inequality for each of our outcomes $\{V, A, D\}$).

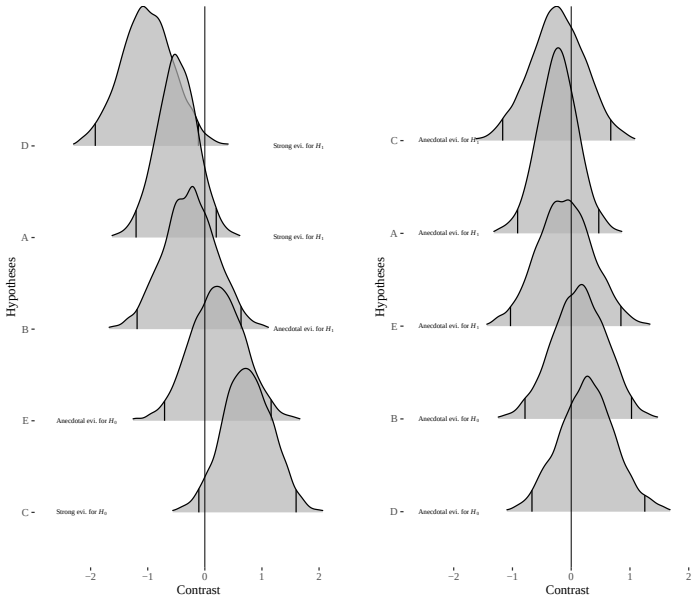
If we plot the posterior probability distributions for each hypothesis test (15 in total), one can more clearly see what a 'significant' effect means in the context (Figs. 11a–11c).

4.1 Effect Sizes

Looking at Figs. 11a–11c one sees three hypotheses that indicate strong evidence, i.e., Examples D, A, C in outcome V (valance). In the two former cases we have *strong* evidence *for* H_1, while in the latter case we have strong evidence *for* H_0. Analyzing the effect sizes for these results is wanted. However, we also see two results that could potentially be of interest also.

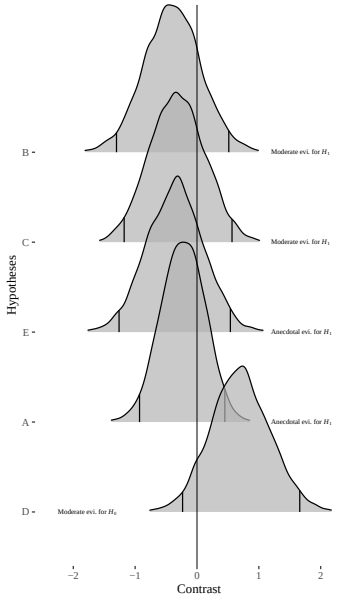
In Fig. 11c, one can see that there are some distributions classified as providing *moderate* evidence for H_1 or H_0, respectively (but they are still fairly close to a quantile). These are Examples B, C, and D. Even though we do not have strong evidence speaking in favor (or not) of a hypothesis, it could be of interest to see what this entails concerning effect size.

We will provide raw effect sizes on the outcome scale since a translation to, e.g., Cohen's d, would not be meaningful when working on Likert scales. In short, we want to see, on average, how large an effect size would be to move from H to L



(a) Valence (V)

(b) Arousal (A)



(c) Dominance (D)

Fig. 11 An overview of all hypothesis tests, given our three outcomes $\{V, A, D\}$. On the y-axis, Example (A–D) is ordered according to the direction of evidence starting with the most negative direction. Next to each distribution, a short note clarifies the results of the tests (according to Table 6). Finally, the distributions have 2.5% and 97.5% quantiles drawn in the tails.

Outcome	Example	Min.	1st quant.	Median	3rd quant.	Max.
Valance (V)	D	-3.6	-1.5	-1.0	1.8	3.9
Valance (V)	A	-3.0	-1.0	-0.5	0.0	2.0
Valance (V)	C	-1.6	0.2	0.8	1.3	3.8
Dominance (D)	B	-3.2	-1.0	-0.4	0.1	2.7
Dominance (D)	C	-3.1	-0.9	-0.3	0.2	2.6
Dominance (D)	D	-2.5	0.2	0.78	1.3	3.8

Table 7 Raw effect sizes from posterior samples ($10,000$ draws) of the posterior predictive distribution. These samples have higher variance than samples of the means of the posterior predictive distribution since residual error is incorporated. The first three rows present raw effect sizes where the hypothesis test found strong evidence, while the last three rows where there was moderate evidence. The median column is the size of the effect (on the outcome scale) for the contrasts L–H. If we have a look at the first row we see an effect size of -1.0, i.e., the difference between Low–High, for Outcome V and Example D, is -1.0 on the Likert scale with the quantiles $[-1.5, 1.8]$. This should not be confused with the hypothesis tests we conducted (Figs. 11a–11c), which tested if Low<High.

for each of the six Examples. By drawing samples from our posterior probability distribution, we can easily compare the difference between levels. We leave all variables according to what we have in the sample (e.g., the distribution concerning Experience is the same), and only vary the Example level to see what this means on the outcome scale. Table 7 provides us with an overview of the six effect sizes.

One can conclude this section by claiming that we have some interesting effects, some even based on substantial evidence. These are summarized in the box below.

Findings for RQ1:

- Cyclically-dependent modularization (ScD-H) is less pleasant than its refactored (ScD-L) counterpart (strong evidence).
- Missing encapsulation (ScA-H) is less pleasant than its refactored (ScA-L) counterpart (strong evidence).
- Broken modularization (ScC-H) is *more* pleasant than its refactored (ScC-L) counterpart (strong evidence).
- Missing Hierarchy (ScB-H) is, likely, *less* dominating than its refactored (ScB-L) counterpart (moderate evidence).
- Broken modularization (ScC-H) is, likely, *less* dominating than its refactored (ScC-L) counterpart (moderate evidence).
- Cyclically-dependent modularization (ScD-H) is, likely, more dominating than its refactored (ScD-L) counterpart (moderate evidence).

Findings for RQ2:

- Work experience, likely, correlates with submissiveness (moderate evidence).

Additional findings:

- Refactored Missing Hierarchy (ScB-L) yielded particularly submissive responses.
- Missing Hierarchy (ScB-H) yielded particularly displeasing responses.
- Refactored Cyclically-Dependent Modularization (ScD-L) yielded particularly pleasing responses.
- Refactored Broken Modularization (ScC-L) yielded particularly displeasing responses.

5 Qualitative Analysis and Results

Analyzing the data set revealed that the participants have strong and negative affects towards TD and are inclined to talk about their reactions. Their argumentation was clearly of the stimulus-response variety, i.e., they viewed TD as an action they are exposed to, leading to counteractions. The participants' discussions centered around what one might think of as defense or coping mechanisms for said stimulus.[6]

The thematic map (two themes and five sub-themes) constructed during the analysis is included in Figure 12. The first theme (three sub-themes) describes the

[6] These are established terms within psychology, and the surrounding theory could not be delved into for the scope of this study. In this article, we will instead use the term *psychological rebound* to avoid overloading the terms.

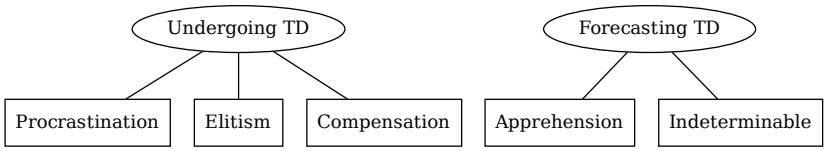


Fig. 12 Thematic map of how software practitioners reason about TD in tandem with affects.

participants' reflections (with regards to affective state) on undergoing TD intense areas (*Undergoing TD*), e.g., encountering TD, when working with some other task.

Among its sub-themes, we first consider *Procrastination*. At its core, this sub-theme is about instances where practitioners try to delay or avoid dealing with the debt or its consequences. Often, this is related to the sense of feeling overwhelmed when facing TD.

Procrastination may surface in several different forms. For example, one interviewee reported that TD could cause task abandonment. *"the more, like, bad code I see [in the same place], the more, like, bored and [indifference] [. . .] It's like, '[vocable of quitting], I give up'. It's like, 'it's too much now, I give up.'"*

This feeling of resignation was echoed by another practitioner, who also suggests that tightly coupled code is cognitively taxing. *"it had this* `instanceof` *bit that implies that it knows about something else, so then you have to start knowing about two places at once, in parallel, and that usually gets super messy. [vocable of distaste] Yeah, so it's, sort of, being in control and being able to fix it."*

At the same time, *Procrastination* is not constrained to low levels of arousal. Quite the opposite, in some instances, it can lead to an impulsive and risky overhaul of parts of the codebase: *"I would throw away and rewrite it"*.

From these examples, it is clear that TD can cause *psychological rebound* effects that are harmful to the software project in ways that go far beyond the human aspects perspective. For example, abandoning tasks because of TD can upset backlog prioritization or result in project slippage. Similarly, the urge to overhaul the codebase could, e.g., invalidate the debt's principal or overrule trade-off analyses.

Unsurprisingly, the participants were aware of the consequences and severity of *Procrastination*. One interviewee said, *"I think the detrimental part is when you feel like you don't wanna touch it [. . .] even if I do touch it in the end, it will take a longer time before I actually dare."*

Next, the second sub-theme is *Elitism*. It encompasses reactions to TD, violating some expectations that one holds oneself, one's colleagues, or the codebase to. In the case of *Elitism*, these exceptions typically do not represent a shared set of

values and beliefs among the parties. Hence, the discourse in this sub-theme was notably flavored by negative interpersonal dynamics.

Elitism is reflected in several different affects that appear to fall on a wide scale of blaming the author of the code. One example of a low amount of blame was one participant who expressed disappointment. *" if you have a great design, a great architecture, following the SOLID principles. That are loosely coupled. [Then,] they [code problems] are easy to fix. The problems. Easy to change. That is the most important, to me. So there are some—they are fundamentals of how I think when I design a program. So [code] violating those principles make me feel very sad."*

As can be seen, this suggests that the participant's affects were influenced by the codebase itself, i.e., more or less decoupled from its author. On the other hand, another interviewee, who experienced distrust, accentuates the author's (perceived) skill and does not separate it from the quality of the code: *"I've seen things where people mix really bad indentation, combined with not having, like, opening and closing brackets for* `for`*-loops, for example. Using, like, short notation. We can have, like, one-liners after* `if`*-statements, for example. I mean, those things are just terrible, 'cause you don't know what belongs where. It's messy and there are, like, no, like, blank lines between—additional spacing between things or anything. It's just a bunch of code, with wrong indentation. Sometimes indented, sometimes not. And unclear what belongs to which statements. [. . .] it's easier to spot it [than architecture]. And it's so, like, something I really think people should know how to do. It's so basic, in programming. So, yeah, I think so. It makes me a bit more worried, so to say, when I see that stuff. 'Cause it's very much easier to do correctly."*

Continuing on this blame scale, examples arise where the code is de-empathized in favor of focusing on its author. For instance, one interviewee expressed scorn and a notion of coding style reflecting one's personality. *"I get a bit annoyed with people that try to be too smart with the programming language. They know, like, a short way of writing things, and they know exactly what happens. [. . .] So I'm more for, like, writing simple, easy to understand code. So that everyone that follows, can easily make changes to it. So yeah, that annoys me a bit: when people try to be too clever. They wanna show off that they are smart, by using, like, weird functions of a language."*

Viewed together, these examples suggest that *Elitism* may arise from misalignment of quality expectations. However, this is perhaps not obvious to the practitioners, as the focus is not on addressing these alignment problems. Clearly, *Elitism* threatens to cause conflict among employees, but can also rationalize TD by acknowledging the debt as a result of business constraints: *"However, I also feel that when I read someone else's code, that's really bad—or shit, or something—I also realize that this might have been done under pressure, depending on the project and stuff. So I accept these technical debts better. Unless it's just plain bad and not time-saving at all."*

Elitism can be dangerous also when it does not result in (external) conflict. One interviewee highlights the risk of it causing high levels of stress. *"Yeah, this was people that were sort of in the more, like, architect roles, usually. Then they put on too much work on their shoulders. They were the guys that always wanted to do everything*

by themselves. And, sort of, tended to burn out after a while, 'cause they just had too much to do. You could see that they were stressed about it [soft deadlines]."

The final sub-theme of the first theme is *Compensation*, which concerns constructively addressing TD. Often, the TD items are viewed as opportunities for improvement. As one interviewee put it, *"So, there definitely is this scope for improvement, but I would not call anybody else's code as poor. [. . .] I generally do not get any negative feelings about it [code clones]. But I do look at it as an opportunity to improve the code myself."*

Compensation is not limited to correcting an instance of TD, but can be preventive actions, e.g., informing the code author about their mistake: *"Personally, I would use* `git blame` *to see who wrote the code and then, if I can contact them, I say 'okay, next time, you should do it better. Because this, like, it may take a lot of time for others to trace their issues.'"*

One interviewee even suggested that affects can be leveraged to improve the code base, as they can act as software quality proxies: *"emotions aren't bad or good. If a team member is that mad about something, I just use that as an indicator that something is bad in the code. So that person is right to be angry, and we can use that to either fix it, or use that as an argument for—like, in the future—like, let's refactor this in the next sprint, or whatever."*

Together, these extracts show that *Compensation* is related to TD management and, more specifically, tactics for addressing TD maturely or constructively. Please note how these tactics are concerned with the practitioners' dominance concerning the codebase. As one participant said, *"I want to rewrite it [code with inheritance issues]. [. . .] to improve it and to just, yeah, maybe so I don't feel stressed about it. So I have control"*

So far, we have presented the components of the first theme. Before continuing with the next theme, the interactions between these components should be analyzed. Note how all three sub-themes appear to be *psychological rebounds* for TD, albeit as different manifestations. *Procrastination* looks like an impulsive and naive, almost childish reaction to TD, where the practitioner does not acknowledge the consequences of their actions. These traits can also be seen in *Elitism*, but with regards to collaboration and teamwork rather than how the debt itself is approached. On the other hand, *Compensation* appears to be a manifestation of thoughtful consideration of how to manage the TD.

The second theme (two sub-themes) describes the participants' reasoning (with regards to affective state) when forecasting the consequences of TD (*Forecasting TD*), e.g., the effects of leaving TD unaddressed.

Its first sub-theme is *Apprehension*, which includes the anxiety of expecting future maintainability issues. A significant part of this sub-theme constitutes the participants' concerns about the extra psychological toll caused by TD. This toll can emerge when the practitioner believes there is a risk of the code leading to system failure. As one participant said, *"The spontaneous feeling was a bit stress about too much stuff going on. Too many components, and some strange dependencies. And too*

much inheritance. [. . .] Why [do I feel stressed]? 'Cause I can see myself maintaining that code. And I can see that code breaking in the long term. [. . .] 'Cause I don't want the system to break."

This kind of uncertainty was echoed by another interviewee, who emphasized the toll of unforeseen consequences (ripple effects): *"for me, it comes back to, like, the control. I know that if I'm gonna touch this, I'm gonna pull a string, and then there's gonna come, like, a spider web with a spider in it. [. . .] You know that when you do something here, it's gonna affect something else."*

However, *Apprehension* is not limited to the technical considerations, as the psychological toll can also appear in the presence of tight schedules. As one practitioner put it, *"If you have time pressure to do something, and then you also know that you're in—I mean, "this is gonna be hard to test. And to deliver it in time is gonna be tough." Then it's super stressful. But if you don't have that pressure again, then it's easier again."*

Clearly, *Apprehension* is found in situations where the practitioner's dominance is on the submissive part of the scale, where they have low confidence in the code. Further, the extracts suggest that work tasks and business considerations are difficult to separate from their affective states. As one participant said, *"I mean, they [the technical and emotional viewpoints] are connected somehow. But through my years—my experience—I see a lot of problems with code violating these [SOLID design] principles. And that causes frustration when you try to fix bugs, improving the code, extend the code. So, it's more from a technical perspective, but they cause negative emotions."*

The last sub-theme is *Indeterminable*, which encompasses the difficulty of decoding TD. That is, understanding or sharing one's understanding of the TD in the system appears to be a non-trivial matter, which could play a key role in valorizing TD items.

In industry, TD items are sometimes so opaque that professionals may not recognize them until they have paid a significant amount of interest. As one interviewee said, *"one time I was just gonna write some test for a thing we did. Then I realized the whole thing was such a debt-cluster that I just had to throw it away. I spent like three, four hours trying to help my team out. I didn't realize I did zero value [laughs] with that time."*

At the same time, TD might be widespread in the system, becoming a sort of background noise challenging to pinpoint. As one participant put it, *"sometimes you actually encounter some area that makes you really unhappy to be in. But then you also have these overarching stuff, that isn't really bothering you that much. But you always knows it. You know it's always there. So it's way—it's less tangible. I would say it's, like, hard to identify. Hard to measure."*

Further, practitioners may recall areas with a high amount of TD but are sometimes unable or unwilling to articulate the problem constructively: *"I hear about '[vocable of complaint] this shitty part of the system'."*

These extracts tell us that software practitioners have trouble estimating and communicating the consequences of existing TD items. However, as suggested by one interviewee, they may hold strong intuitions. *" In industry, it's more 'I know something is wrong. It feels like things are spread out like this. I just can't put my finger on it.' [. . .] The feeling I have in industry is more like, 'I know I'm gonna work in this area. I know it's gonna be horrible. I don't know what's gonna happen, exactly. Something is gonna show up. It's gonna take longer time. I can't give you a real estimate for how much it is to fix all of it, and I can't give you a real business value, because I just know it's gonna be hell.'"*

In conclusion, the analysis reveals that affects are very much a key aspect of TD. They provide an insight into the underlying mechanics for how software practitioners respond to TD items. These *psychological rebounds* may be a necessary consequence of TD and should not be ignored. The findings are further summarized in the following data extract and the box below. *" if it's [the debt is] manageable or if I feel like I can fix it, then it feels a bit okay. It's like, 'oh, this is a crappy thing someone did, but—whatever, it's fixable' in contrast to, like, 'this is just a nest of—we just need to re-engineer.' That makes you just angry inside. [. . .] you can definitely feel when it's '[vocable of excitement], I can refactor this' or [. . .][vocable of quitting], this is such a mess. I hate going into this code. I can't fix a bug here, 'cause there's just going to pop up things in other places. So, it's a mix. Depends on how much impact you can have on it, I think. Because it can be really fun to actually fix stuff. But when you can't, then it's like '[vocable of annoyance], angry.'"*

Findings for RQ3:

- Software practitioners experience (strong) affects from TD along all three dimensions.
- When faced with high (overwhelming) levels of TD, practitioners will be reluctant to perform their work tasks.
- Time pressure is sometimes a catalyst for negative affects.
- Viewing TD items as opportunities for improvement appears to correlate with dominance towards the codebase.
- TD anxiety relates to code dependencies, ripple effects, and (the risk of) defect introduction.
- TD anxiety appears to be correlated with submissiveness towards the codebase.
- Displeasure plays an important role in recognizing the presence and severity of TD.
- Software practitioners sometimes get positive affects from amortizing TD.
- Profanity frequently emerges in TD discussions.

Additional findings:

- Quality processes sometimes get disrupted by software practitioners' affects.
- Misalignment of quality expectations may result in interpersonal conflicts or burnout.
- TD is challenging to decode (recognize, estimate, and communicate).
- Violations of something the software practitioner considers fundamental appears to result in stronger affects.

6 Discussion

In this section, we tie together the results from the quantitative and qualitative analyses. First, we discuss the quantitative results related to the various scenarios and smells; to better explain the results, we then explore the quotes from the qualitative data occurring in correspondence with the analyzed smells. This allows us to explain how our results answer RQ1, or else how the smells influence the participants' affects. We compile a ranked list of which smells seem to have more impact.

We also discuss how changes in affective state align with professional characteristics (RQ2). We then take a broader scope and reason on the exploratory results from the qualitative analysis and what relationships we have found between affective states and technical debt (RQ3).

6.1 Case A: Missing Encapsulation

The quantitative data strongly suggests that the presence of the smell related to missing encapsulation in the code (ScA-H) causes the software practitioners to feel less pleasure (valence). This entails that practitioners consider the presence of this smell with disapproval rather than with indifference.

We do not seem to find other significant evidence related to the other two dimensions (arousal and dominance), which could imply that the practitioners do not consider this smell exceedingly threatening. This is also mentioned in the qualitative data, as one of the participants mentioned: *"Like, some of them were quite [vocable of annoyance] as solutions, but didn't really impact me that much. Like, the rectangle whatever—PNG-things [reference to ScA-H]. Like, yeah, I can refactor this in an afternoon."*

On the other hand, such a lack of strong feelings could be caused by the limited size and localization of the example and how easy it is to estimate the practitioner's refactoring. One of the interviewees mentioned *"So, it's—this, like, rectangle-PNG-thing [reference to ScA-H]—it's, like, I can really point to it. Show it. I can give an estimate for how much time is left and how much impact it is. The feeling I have in industry is more like, 'I know I'm gonna work in this area, I know it's gonna be horrible. I don't know what's gonna happen, exactly.'"*

In conclusion, the smell is recognized as a problem, but not as a high-priority one. Suppose we consider the strength of the resulting feelings and the participants' insights for this smell. In that case, we can conclude that the presence of this smell, although frowned upon, is perhaps not considered detrimental by the software practitioners.

6.2 Case B: Missing Hierarchy

We did not find evidence to support the hypothesis that this smell generates any negative feeling in the software practitioners. Surprisingly, on the contrary, we found (moderate) evidence that practitioners felt more dominant (dominance) in working with the code containing the smell (ScB-H).

On the other hand, this scenario was mentioned a lot in the qualitative analysis in rather negative terms. However, those comments often referred to the whole code and not to the specific smell. Although, at the same time, some participants explicitly mentioned the smell and suggested the correct refactoring. *"Yeah, one example, that had the private class there [referring to ScB-H], and that one I didn't like [...] Yeah, overall the checking of types in code like that: I think it's a sign of bad architecture, most of the time, when you have to check the type of objects coming in. Then you can probably—yeah, like I said—interface it out. And an interface coming in and you have the method on the interface and, yeah."*

The scenario itself could explain these seemingly contradictory results: debt-intense areas could bias the practitioner to distrust suitable constructs. After all, understanding and implementing a correct hierarchy involves a greater ability of abstraction. In other words, given that the original developers fell short in performing more straightforward tasks, confidence would be low to succeed in more demanding activities. As one of the interviewees said, in a different context, *"it's so, like, something I really think people should know how to do. It's so basic, in programming. So, yeah, I think so. It makes me a bit more worried, so to say, when I see that stuff. 'Cause it's very much easier to do correctly."*

Another, less plausible explanation would be that practitioners feel submissive (intimidated) *because* of the necessary abstraction skills, i.e., are not comfortable with such constructs. Here, it is essential to note the gap between recognizing a suspicious programming language construct (`instanceof`) and intimately understand which abstraction would be suitable. The former is a low-level pattern detected by static code analysis (or even text search), while the latter often requires domain knowledge, experience, and other unquantifiables. However, this seems unlikely, as most participants were experienced in object-oriented programming languages.

Finally, a third explanation could be that abstractions (by definition) remove details from the context. In other words, while beneficial for the system's maintainability, abstractions might, locally, result in less insight and, hence, less control.

In summary, the findings suggest that the smell is considered a problem despite its positive impact on dominance. The surrounding code's quality appears to confound individual TD items, but this effect needs to be verified in future studies.

6.3 Case C: Broken Modularization

Similarly to Case B, we did not find evidence to support the hypothesis that this smell generates any negative feelings in the practitioners. However, we did find (strong evidence) that they felt more pleasure (valence) and (moderate evidence) more in-control (dominance) when working with the code containing the smell (ScC-H).

This can be explained by the fact that the broken modularization smell consists of a widely recognized correct approach (modularization) applied in the wrong way (broken). In particular, the code that was modularized did not need to be (it consists of just variable declarations), and it should have been contained in the same abstraction (ScC-L). However, the participants' feelings might have been triggered by the presence of a better visual structure in ScC-H. The lack of a counter-effect for the displacement of the modularized code (ScC-H) could have different implications:

1) The positive feelings in the presence of modularized code far outperforms the negative feelings related to the sub-optimal use of such mechanism. This is also supported by one of the participants: *"It's, like, I could sort of see what had*

happened, I think. Like, the last one [reference to ScC-L] with the weird device. It looked like a container of data and someone plonked helper methods in it, maybe, I don't know. It's, like, I can see how that happened. I can move them without changing anything, so there won't be any ripple effects and I can still improve the code, for instance."

2) The practitioners could have overlooked the specific code that was modularized in an additional class, focusing more on the structure rather than on the code itself. Alternatively, the participants might have thought that the additional class, the results of the modularization (which contains only variable declarations in our example), could have contained additional methods that were not displayed in our snippet.
3) While modularization is a well-known good practice, broken modularization is a less well-known bad practice among the practitioners. This can also be related to the language used by the participants and their familiarity with object-oriented programming. However, we did not find any evidence in the quantitative results supporting such an explanation.

In summary, we could not find evidence that this smell generates negative feelings in software practitioners. On the contrary, it seems as though the code with the smell was liked more, probably because the participants did not recognize (consciously or unconsciously) the misuse of modularization as significantly impacting.

6.4 Case D: Cyclic Dependencies

This is the smell for which we have quite strong evidence supporting hypotheses from literature (Martini et al., 2018b; Al-Mutawa et al., 2014). We can see how, for valence, the software practitioners reported extra-pleasure in the presence of code that is refactored (ScD-L), while at the same time, we register strong evidence that such code is much better liked than the one with the smell (ScD-H). Our analysis also reports moderate evidence for dominance, where practitioners feel much more in control of refactored code (ScD-L) than the code containing the smell (ScD-H).

Despite such strong results, the smell was not often or explicitly mentioned in the qualitative answers of the practitioners dealing with ScD-H, if not for the two quotes below, which can be related to this specific smell: *"The spontaneous feeling was a bit stress about too much stuff going on. Too many components. And some strange dependencies."* and *"the code doesn't have to be perfect, or there could be some problems with the code. But if you have a great design, a great architecture, following the SOLID principles. That are loosely coupled. They are easy to fix."* This could have happened because other smells or scenarios were more interesting to discuss, either because this example was not considered too challenging (perhaps because of the limited size of the example) or, possibly but perhaps less likely, because it was more noticeable and therefore less interesting.

In summary, we can consider this as evidence that the presence of cyclic dependencies generates stronger negative feelings in practitioners along at least two dimensions (valence and dominance).

Although this can be somewhat expected (cyclic dependencies is a well-known smell, probably more than the other smells), it is interesting to note how the degree of negative feelings for this smell far exceeds other smells. We find this plausible: cyclic dependencies is the smell that tends to involve multiple entities (usually classes), which can generate ripple effects across the code. Also, the example that we propose here consists of just one dependency. However, the cyclic dependencies anti-pattern could consist of several dependencies and several involved entities, which would increase any harmful effect. Suppose one were to project the impact that we have recorded on the affective state in this simple example for a larger one. In that case, we could expect even stronger negative feelings affecting practitioners at the sight of this smell. On the other hand, this was a clear and obvious case of cyclic dependency. In contrast, such dependencies, especially if involving several entities, can become less noticeable and not so visible if they are not explicitly investigated, as shown in other publications, see, e.g., Martini et al. (2018b) and Al-Mutawa et al. (2014).

6.5 Case E: Rebellious Hierarchy

We did not find even moderate evidence that this smell would generate either positive or negative feelings in the participants concerning any of the dimensions. Therefore, it is difficult to draw conclusions on this smell and its impact on the practitioners' affects. In general, it seems as the participants would be quite indifferent to one or the other proposed solution. For example, even when encountering ScE-L, one of the practitioners mentioned: *"you had the* `Document` *[reference to ScE-L], yeah that was the wrong structure of the abstract class, I think, because you had all those methods, but only some of the implementation used. They didn't represent the same object. If you looked in the implementation, they had different actions or abilities. I think the public part of the implementations should be the same."*

The lack of evidence in itself combined with the quotation could point to three possible conclusions:

1) This smell is not considered a problem by practitioners, and it does not affect them.
2) Our example was not a good representation of the actual issue. Unfortunately, we did not find an existing implementation that would suit our experiment, so we had to adapt our snippet from Suryanarayana et al. (2014), removing any domain-specific reference that our participants would not understand. This process might have excessively simplified the smell.
3) The smell consisted of one short method out of ten, distributed in four (ScE-H) and five (ScE-L) classes, respectively. This could mean that the participants might have overlooked it in the time allowed for the task, perhaps focusing their attention on other code features, such as its structure (as mentioned in the previous quotation from a participant).

Smell	Impact on affects	Dimensions	Other considerations
D – Cyclic dependencies	Negative Negative (likely)	Valence Dominance	Not explicitly discussed qualitatively
A – Missing encapsulation	Negative	Valence	Easy to estimate a refactoring
B – Missing hierarchy	Positive (likely)	Dominance	Recognized as bad practice, but overshadowed by the scenario code
E – Rebellious hierarchy	-	-	Seems to not have been recognized as an issue
C – Broken modularization	Positive Positive (likely)	Valence Dominance	Modularization seems to give positive feelings even if misused

Table 8 Prioritized smells according to our findings.

In conclusion, we cannot draw much conclusions from these results, although we can speculate that the participants have not recognized this as an issue affecting their feelings.

6.6 Comparison Across the Smells

We have so far reported our reflections, based on available evidence, on how and why the different smells have impacted the participants' affects. However, can we say something more about how the different smells compare to each other?

We report a summary (see Table 8), in which we compile a prioritized list of the smells based on the reported evidence. The smells are ordered by their negative impact on the software practitioners' affects. We also report if other impacts have been found on the refactored solution. To clarify the results, we have arranged the relationship positive/negative concerning the quantitative analysis, i.e., we do *not* consider the direction of the SAM. For example, in Table 8 'negative impact on valence' means displeasure. We also highlight the strength and type of evidence supporting our conclusions.

6.7 Suggestions for Practitioners

Based on Table 8, we can undoubtedly suggest practitioners pay attention, especially to cyclic dependencies and missing encapsulations. As for the latter one, the practitioners mention that its refactoring would not be costly, which could make it a good candidate for a mandatory cleanup of the code before release.

We also, less vigorously, suggest that practitioners keep their eyes out for missing hierarchies. While the presence of this smell *increased* dominance that could be

indicative of other problems, e.g., a need for domain knowledge acquisition. In particular, practitioners should consider taking action if developers start introducing this smell despite them recognizing it as bad practice.

As for the rebellious hierarchy, the study results do not allow us to draw firm conclusions, maybe because such smells might not be considered so upsetting by the participants. Finally, and probably surprising, it seems that a modularization that is not entirely correct is not considered problematic. At the same time, developers might tend to consider the pleasure of modularized code (even if containing a smell) better than smell-free code, which might be less modularized.

However, we need to notice that, for missing encapsulation, rebellious hierarchy, and parts of broken modularization, the conclusions cannot be considered very strong, as our evidence is moderate or lacking. These results are also related to the influence of the smells on the participants' affects and do not consider other negative or positive effects. However, we consider our findings important to report, as (see Section 2.3) developer unhappiness has been linked to harmful consequences. Further, the results should not be confused with the actual extra-maintenance effect these smells have in practice, although the two variables are most probably correlated.

6.8 The Effect of Experience

Our quantitative results do not point to many correlations between the participants' professional characteristics and how the affective states changed. The most striking results are related to the experience of the respondents.

Experience has shown (moderate evidence) to have a negative impact on Dominance. In other words, the more work experience the subject had, the more submissiveness they report.

A possible explanation for the increased submissiveness is that more experienced practitioners have dealt with the technical debt related to the smells for a longer time than junior ones. This may be caused by the fact that they have witnessed more of the technical debt's long-term negative impact, which may trigger the additional caution for the smells (Dunning–Kruger effect).

These results also seem in line with our qualitative findings, primarily related to the maturity (see next section) with which practitioners undergo TD: we could argue that, with less experience (and, probably, less maturity), practitioners seem to want to ignore TD and avoid to worry about it (as highlighted by the *Procrastination* theme), hence the presence of lower submissiveness. Then, moving to a more elitist and compensating attitude towards TD as they gain more experience, they become additionally worried when encountering TD (as shown in the feeling of lacking control, mentioned in relation to the *Apprehension* theme).

Additionally, these results are in line with what is reported concerning how startup teams are composed and their inclination to incur TD. Besker et al. (2018b) report on interviewees from startups mentioning how it can be considered better to include a large part of junior developers in the initial team, to make sure that TD is accrued (saving resources in the face of an initial high risk of failure). Experienced practitioners (apart from a small initial fraction) would be more suited for the growth and mature phase of a startup when TD needs to be removed before it becomes disruptive.

6.9 The Overall Effect of TD on Affects

Participants report that TD items activate a substantial portion of the emotional room (three dimensions), including vivid ones (e.g., profanity occurred). Still, our experiment showed nothing concerning the arousal dimension. A plausible explanation is that participating in the experiment represents a different situation than encountering technical debt in real projects. The technical debt encountered during the experiment is not directly and negatively impacting the practitioners with, for example, extra-effort or additional bugs. This means that the arousal dimension could be triggered in a different context.

Many participants receive satisfaction from improving code. Mainly, being able to perceive their work as impactful causes pleasure. On the contrary, the uncertainty caused by code affected by TD and the consequent distrust in the codebase are sources of negative feelings. Architectural TD is considered a common source of negative feelings, especially for problems related to ripple effects (as, for example, in case *D* for cycling dependencies).

Then the question is: why is TD so present in industry, and why is, e.g., code not continuously refactored?

First, as practitioners reported, stress is prevalent in the software industry. Several participants see deadlines as negatively affecting themselves and the product. Avoiding TD requires more time, which would increase the stress in the presence of a deadline. This might mean that practitioners, to avoid stress, prefer to incur TD. The participants also mention that TD problems encountered in their daily lives are more extensive and more obscured than those in the experiment. This most probably hinders them from being able to fix TD issues efficiently.

Another point of consideration was raised in the qualitative analysis, namely, that each sub-theme for undergoing TD is a *psychological rebound*. Further, there seems to be a sort of progression to them, which we will refer to as *maturity*, as we can draw parallels to our previous experience with group development models (Gren et al., 2017).

First, *Procrastination* ("Forming") can be interpreted as a mechanism with little interest in improving the situation. Consequently, the practitioner will not attempt to share the team's burden nor attempt to shield its members from the harmful

stimulus. Next, *Elitism* ("Storming") involves questioning the codebase and the *modus operandi*, which can be destructive and socially taxing unless adequately managed. Finally (note the absence of "Norming") *Compensation* ("Performing"), illustrates a successful transition from defensive reactions to coping ones, with the participants focusing on facing up to the TD item and resolving it constructively.

7 Limitations and Threats to Validity

Conducting empirical studies with human subjects is often a complex issue (Miller, 2008), as it is often the case that the context is noisy and investigated effects are small (Gelman, 2018). As such, the potential threats to validity are often numerous, and it would be infeasible to discuss them all fully. This section presents what we consider the most significant validity threats to this study and the measures taken to mitigate them. The threats are categorized according to the aspects suggested by Wohlin et al. (2012) and Runeson and Höst (2009).

7.1 Construct Validity

This study set out with the aim of determining how DTD relates to the affects of software practitioners. One of the methods used was a repeated-measures experiment, where the participants were presented with five software design smells and their respective refactored versions. However, those smells were instantiated in code examples that originated from the same source, namely (Suryanarayana et al., 2014), which is not a scientific publication (although a derivative of one). Also, the source's purpose differs from ours in that the smells and the refactored versions are intended to be contrasted with each other. As previously stated, the rationale of this choice was a perceived deficit of concrete DTD representations in the research literature (for our purposes, at least).

These characteristics introduce threats to validity, but because they were identified before the data collection, countermeasures could be introduced. Since we were unable to find a way to eliminate these threats, we chose to monitor them and investigate the issue post-hoc. This was done by introducing validity-checking questions (see Table 3, questions IQ4.1, IQ4.2, and IQ5) to the interview questions and analyzing the answers. (Further, IQ3, IQ6, and IQ7 checked other types of validity.)

The participants confirmed that the scenarios were representative industry code, albeit atypically small and isolated examples of TD encountered in practice. This suggests that the treatment was suitable and that the resulting data is an *underestimation* of the industry's situation.

7.2 Internal Validity

The laboratory experiment part of this study was a repeated-measures design. While this approach lowers the threat of confounders, because each subject's peculiarities are accounted for, it is more susceptible to learning effects.

Several countermeasures were taken in order to reduce learning effects. First, the participants were acquainted with the situation during the first phase of the session (pre-task instructions), and each received the same instructions for how to use the measurement instrument and their task. Second, the participants obtained practical experience with the procedure before the measurements (anchor point). Third, intermissions were used between measurements (deacclimatization periods) to lower the probability of any affects induced by previous scenarios carried over to later ones. Fourth, each participant was randomly allocated to one of two complementary treatment patterns designed to minimize bias. Fifth, and perhaps most importantly, *the order of the scenarios* was randomized for each participant.

7.3 External Validity

For this study, its worth noting that the field of psychology is experiencing a replication crisis. Unfortunately, we have been unable to find consensus on concrete best practices for ensuring replicability and have instead chosen to adopt some propositions.

We have made our work as transparent as possible, under the constraints set by confidentiality, anonymity, and copyright. This means that we, in addition to the statistical analysis results, have published data (replication package). Similarly, we have described the procedure at length and made the experiment material available (except the code examples, due to unanswered request for permission).

Another issue concerned with external validity is the sampling strategy. In this study, we employed convenience sampling (finding companies through our professional networks). The approach meant that the sample was limited in several ways. First, all participants were industry professionals, which is a subset of all software practitioners and might not be representative. Second, the participants were selected by managers, who might have their agenda in what employees to select. Third, the companies belonged to the subset of companies that were both sufficiently interested in this study and could allocate resources (i.e., subjects).

However, our results show that the data is inconclusive concerning the effects of different professional characteristics, such as work experience and role. This could indicate that the study is less susceptible to convenience sampling than otherwise. Further, the 40 participants had a wide variety of professional backgrounds and were employed at eleven different companies and one government agency.

Along the same line, the generalizability of the results of the study is threatened by demographic factors. Due to various constraints (including financial), all partaking entities had offices in Sweden. While the study was conducted in several parts of the country, Sweden is culturally distinguished in terms of secular-rational and self-expression values (Inglehart and Welzel, 2010). That said, there was diversity in, e.g., ethnicity among the participants, but such data was not collected to protect confidentiality. For the same reason, many aspects of the participants' demographic profiles were not investigated.

7.4 Conclusion Validity

The investigation undertaken in this study is novel in the sense that, as far as we can tell, no previous studies have investigated how DTD relates to affects. Consequently, the findings of this study cannot be compared and contrasted with the findings of others. Instead, they must be evaluated in isolation and are, therefore, more susceptible to incorrect inferences and conclusions.

To combat these threats, three triangulation techniques (Miller, 2008) were adopted. First, the data were triangulated in the sense that the sessions were spread out over four weeks, and the participants were employed at different entities. Second, researcher triangulation was achieved as two researchers took part in all data gathering and interpretation. Third, methodological triangulation was used, as data were collected through an experiment, a questionnaire, and interviews.

8 Conclusion

Fully understanding the impact of Technical Debt (TD) in the codebase is a crucial challenge for researchers and practitioners alike. Although previous research highlighted how TD could impact developers' morale, there is scarce evidence on how specific technical debt issues impact practitioners' affective states. Even more challenging is finding evidence related to design and architectural debt.

With our experiment, encompassing a solid quantitative data collection and analysis supported by additional qualitative insights from the participants, we offer a first detailed look into how the presence of design debt issues affect software practitioners' affective state.

The results show that five different smells have different impacts. Even when present in a small example, cyclic dependencies clearly and negatively affect software practitioners' affects. Simultaneously, missing encapsulation seems to be a more straightforward issue to deal with (although mildly affecting the practitioners' affects). Two issues related to hierarchy (missing hierarchy and rebellious hierarchy) seem to have a conflicting or no evident effect on the participants' affective state. In contrast, surprisingly, the presence of the broken modularization issue seems to have a positive impact on practitioners' affects.

From our qualitative findings, it seems that practitioners undergo different levels of maturity in how they deal with TD. First, they might naively tend to avoid it (*Procrastination*), then they tend to build a quality-heavy mindset (mostly, however, by blaming others for the presence of TD, i.e., *Elitism*). Finally, they reach a higher level of maturity when a constructive mindset promotes high-quality code (*Compensation*). Also, practitioners seem to be affected negatively when they forecast TD, especially with *Apprehension* related to the future negative impact generated by TD, and by the inherent difficulty in identifying TD and predicting its consequences (TD as *Indeterminable* items).

Finally, we investigated if participants' background covariates played a role, and we only found how experience seems to act as a sort of amplifier for the participants' feelings, probably due to repeated encounters with TD and to the different maturity, acquired with more experience, in dealing with TD.

In summary, only some of the known issues highlighted in the literature seem to affect practitioners' feelings. At the same time, we find that dealing with TD is stressful and might require a fair amount of experience in the team to be handled constructively.

This topic remains mostly uncharted, and presents many opportunities for future work. A singular study is insufficient to build a solid theory, but we encourage others to replicate our experiment under similar or different settings, e.g., design smells, TD type, or cultures. Two particularly interesting investigations would be using industry code examples and situations that simulate time pressure.

Similarly, we discovered several peripheral and related research topics. Investigations of burnout, concerning TD and release deadline (or, in the case of continuous delivery, lack thereof), would be most welcome. As would further studies on the idea of psychological *maturity* towards TD.

Acknowledgements We would to thank all companies and participants that took part in our study, as well as all students who participated in our pilot studies.

References

Al-Mutawa HA, Dietrich J, Marsland S, McCartin C (2014) On the shape of circular dependencies in java programs. In: 2014 23rd Australian Software Engineering Conference, IEEE, pp 48–57

Alves NSR, Ribeiro LF, Caires V, Mendes TS, Spínola RO (2014) Towards an ontology of terms on technical debt. In: 2014 Sixth International Workshop on Managing Technical Debt, IEEE, pp 1–7

Alves NSR, Mendes TS, de Mendonça MG, Spínola RO, Shull F, Seaman C (2016) Identification and management of technical debt: A systematic mapping study. Information and Software Technology 70:100–121

Ampatzoglou A, Ampatzoglou A, Chatzigeorgiou A, Avgeriou P (2015) The financial aspect of managing technical debt: A systematic literature review. Information and Software Technology 64:52–73

Avgeriou P, Kruchten P, Ozkaya I, Seaman C (2016) Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). Dagstuhl Reports 6(4):110–138, DOI 10.4230/DagRep.6.4.110, URL http://drops.dagstuhl.de/opus/volltexte/2016/6693

Besker T, Martini A, Bosch J (2017) The pricey Bill of Technical Debt-When and by whom will it be paid? In: IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China

Besker T, Martini A, Bosch J (2018a) Managing architectural technical debt: A unified model and systematic literature review. Journal of Systems and Software 135:1–16, DOI 10.1016/j.jss.2017.09.025, URL https://linkinghub.elsevier.com/retrieve/pii/S0164121217302121

Besker T, Martini A, Lokuge RE, Blincoe K, Bosch J (2018b) Embracing technical debt, from a startup company perspective. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 415–425

Besker T, Martini A, Bosch J (2019) Software developer productivity loss due to technical debt—a replication and extension study examining developers' development work. Journal of Systems and Software 156:41–61

Besker T, Ghanbari H, Martini A, Bosch J (2020) The influence of technical debt on software developer morale. Journal of Systems and Software p 110586

Betella A, Verschure PFMJ (2016) The Affective Slider: A Digital Self-Assessment Scale for the Measurement of Human Emotions. PLOS ONE 11(2):e0148037, DOI 10.1371/journal.pone.0148037, URL https://dx.plos.org/10.1371/journal.pone.0148037

Boehm BW, Papaccio PN (1988) Understanding and controlling software costs. IEEE transactions on software engineering 14(10):1462–1477

Bradley MM, Lang PJ (1994) Measuring emotion: the self-assessment manikin and the semantic differential. Journal of behavior therapy and experimental psychiatry 25(1):49–59

Braun V, Clarke V (2006) Using thematic analysis in psychology. Qualitative research in psychology 3(2):77–101

Bürkner PC (2017) brms: An R package for Bayesian multilevel models using Stan. Journal of Statistical Software 80:1–28, DOI 10.18637/jss.v080.i01

Bürkner PC (2018) Advanced Bayesian multilevel modeling with the R package brms. The R Journal 10(1):395–411, DOI 10.32614/RJ-2018-017

Bürkner PC, Vuorre M (2019) Ordinal regression models in psychology: A tutorial. Advances in Methods and Practices in Psychological Science 2(1):77–101, DOI 10.1177/2515245918823199

Cinaz B, Arnrich B, Marca R, Tröster G (2013) Monitoring of mental workload levels during an everyday life office-work scenario. Personal and ubiquitous computing 17(2):229–239

Colomo-Palacios R, Hernández-López A, García-Crespo Á, Soto-Acosta P (2010) A study of emotions in requirements engineering. In: World Summit on Knowledge Society, Springer, pp 1–7

Cruz S, da Silva FQ, Capretz LF (2015) Forty years of research on personality in software engineering: A mapping study. Computers in Human Behavior 46:94–113

Cruzes DS, Dybå T (2011) Recommended steps for thematic synthesis in software engineering. In: 2011 International Symposium on Empirical Software Engineering and Measurement, IEEE, pp 275–284

Cunningham W (1992) The wycash portfolio management system. ACM SIGPLAN OOPS Messenger 4(2):29–30

Cunningham W (2009) Ward Explains Debt Metaphor. URL http://wiki.c2.com/?WardExplainsDebtMetaphor

Ernst NA, Bellomo S, Ozkaya I, Nord RL, Gorton I (2015) Measure it? manage it? ignore it? software practitioners and technical debt. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, pp 50–60

Fagerholm F, Ikonen M, Kettunen P, Münch J, Roto V, Abrahamsson P (2015) Performance alignment work: How software developers experience the continuous adaptation of team performance in lean and agile environments. Information and Software Technology 64:132–147

Feldt R, Angelis L, Torkar R, Samuelsson M (2010) Links between the personalities, views and attitudes of software engineers. Information and Software Technology 52(6):611–624

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

Fontana FA, Pigazzini I, Roveda R, Tamburri D, Zanoni M, Di Nitto E (2017) Arcan: a tool for architectural smells detection. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), IEEE, pp 282–285

Furia CA, Feldt R, Torkar R (2019) Bayesian data analysis in empirical software engineering research. IEEE Transactions on Software Engineering pp 1–1, accepted for publication

Ganesh S, Sharma T, Suryanarayana G (2013) Towards a principle-based classification of structural design smells. J Object Technol 12(2):1–1

Garcia J, Popescu D, Edwards G, Medvidovic N (2009) Toward a catalogue of architectural bad smells. In: International Conference on the Quality of Software Architectures, Springer, pp 146–162

Gelman A (2018) The failure of null hypothesis significance testing when studying incremental changes, and what to do about it. Personality and Social Psychology Bulletin 44(1):16–23

Gelman A, Tuerlinckx F (2000) Type S error ratest for classical and Bayesian single and multiple comparison procedures. Computational Statistics 15:373–390, DOI 10.1007/s001800000040

Gelman A, Hill J, Yajima M (2012) Why we (usually) don't have to worry about multiple comparisons. Journal of Research on Educational Effectiveness 5(2):189–211, DOI 10.1080/19345747.2011.618213

Gomez P, Zimmermann PG, Guttormsen Schär S, Danuser B (2009) Valence lasts longer than arousal: Persistence of induced moods as assessed by psychophysiological measures. Journal of Psychophysiology 23(1):7–17

Goodman SN (1999a) Toward evidence-based medical statistics. 1: The p value fallacy. Annals of Internal Medicine 130(12):995–1004, DOI 10.7326/0003-4819-130-12-199906150-00008

Goodman SN (1999b) Toward evidence-based medical statistics. 2: The Bayes factor. Annals of Internal Medicine 130(12):1005–1013, DOI 10.7326/0003-4819-130-12-199906150-00019

Graziotin D, Wang X, Abrahamsson P (2014) Happy software developers solve problems better: psychological measurements in empirical software engineering. PeerJ 2:e289, DOI 10.7717/peerj.289, URL https://doi.org/10.7717/peerj.

289
Graziotin D, Wang X, Abrahamsson P (2015a) Do feelings matter? On the correlation of affects and the self-assessed productivity in software engineering. Journal of Software: Evolution and Process 27(7):467–487
Graziotin D, Wang X, Abrahamsson P (2015b) The affect of software developers: common misconceptions and measurements. In: Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering, IEEE Press, pp 123–124
Graziotin D, Wang X, Abrahamsson P (2015c) Understanding the affect of developers: theoretical background and guidelines for psychoempirical software engineering. In: Proceedings of the 7th International Workshop on Social Software Engineering, ACM, pp 25–32
Graziotin D, Fagerholm F, Wang X, Abrahamsson P (2017) On the unhappiness of software developers. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, ACM, pp 324–333
Graziotin D, Fagerholm F, Wang X, Abrahamsson P (2018) What happens when software developers are (un) happy. Journal of Systems and Software 140:32–47
Gren L, Torkar R, Feldt R (2017) Group development and group maturity when building agile teams: A qualitative and quantitative investigation at eight large companies. Journal of Systems and Software 124:104–119, DOI 10.1016/j.jss.2016.11.024
Guo Y, Seaman C, Gomes R, Cavalcanti A, Tonin G, Da Silva FQ, Santos AL, Siebra C (2011) Tracking technical debt—an exploratory case study. In: 2011 27th IEEE international conference on software maintenance (ICSM), IEEE, pp 528–531
Inglehart R, Welzel C (2010) Changing mass priorities: The link between modernization and democracy. Perspectives on Politics 8(2):551–567
Khan IA, Brinkman WP, Hierons RM (2011) Do moods affect programmers' debug performance? Cognition, Technology & Work 13(4):245–258
Kruschke JK (2010) What to believe: Bayesian methods for data analysis. Trends in cognitive sciences 14(7):293–300
Lang PJ (1980) Behavioral treatment and bio-behavioral assessment: Computer applications. Technology in mental health care delivery systems pp 119–137, URL http://www.citeulike.org/user/acagamic/article/3756983
Lang PJ, Bradley MM, Cuthbert BN (1997) International affective picture system (IAPS): Technical manual and affective ratings. NIMH Center for the Study of Emotion and Attention 1:39–58
Leek J, McShane BB, Gelman A, Colquhoun D, Nuijten MB, Goodman SN (2017) Five ways to fix statistics. Nature pp 557–559, DOI 10.1038/d41586-017-07522-z
Lenberg P, Feldt R, Wallgren LG (2015) Behavioral software engineering: A definition and systematic literature review. Journal of Systems and Software 107:15–37, DOI 10.1016/j.jss.2015.04.084, URL http://www.sciencedirect.com/science/article/pii/S0164121215000989
Li Z, Avgeriou P, Liang P (2015) A systematic mapping study on technical debt and its management. Journal of Systems and Software 101:193–220
Lim E, Taksande N, Seaman C (2012) A balancing act: What software practitioners have to say about technical debt. IEEE software 29(6):22–27
Martini A, Bosch J (2017) On the interest of architectural technical debt: Uncovering the contagious debt phenomenon. Journal of Software: Evolution and

Process 29(10):e1877
Martini A, Besker T, Bosch J (2018a) Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. Science of Computer Programming 163:42–61
Martini A, Fontana FA, Biaggi A, Roveda R (2018b) Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company. In: European Conference on Software Architecture, Springer, pp 320–335
Miller J (2008) Triangulation as a basis for knowledge discovery in software engineering. Empirical Software Engineering 13(2):223–228
Morris JD (1995) Observations: SAM: The self-assessment manikin: An efficient cross-cultural measurement of emotional response. Journal of Advertising Research
Morris JD, Woo C, Geason JA, Kim J (2002) The power of affect: Predicting intention. Journal of Advertising Research 42(3):7–17
R Core Team (2019) R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, URL https://www.r-project.org/
R Core Team (2020) R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria, URL https://www.R-project.org/
Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empirical software engineering 14(2):131
Russell JA, Mehrabian A (1977) Evidence for a three-factor theory of emotions. Journal of Research in Personality 11(3):273–294, DOI 10.1016/0092-6566(77)90037-X
Sharma T, Spinellis D (2018) A survey on software smells. Journal of Systems and Software 138:158–173
Spínola RO, Vetrò A, Zazworka N, Seaman C, Shull F (2013) Investigating technical debt folklore: Shedding some light on technical debt opinion. In: 2013 4th International Workshop on Managing Technical Debt (MTD), pp 1–7, DOI 10.1109/MTD.2013.6608671
Stan Development Team (2020) RStan: the R interface to Stan. URL http://mc-stan.org/, r package version 2.21.1
Suryanarayana G, Samarthyam G, Sharma T (2014) Refactoring for Software Design Smells: Managing Technical Debt. Morgan Kaufmann
Tamburri DA, Kruchten P, Lago P, van Vliet H (2013) What is social debt in software engineering? In: 2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), IEEE, pp 93–96
Tom E, Aurum A, Vidgen R (2013) An exploration of technical debt. Journal of Systems and Software 86(6):1498–1516, DOI https://doi.org/10.1016/j.jss.2012.12.052, URL http://www.sciencedirect.com/science/article/pii/S0164121213000022
Vehtari A, Gelman A, Gabry J (2017) Practical bayesian model evaluation using leave-one-out cross-validation and waic. Statistics and Computing 27:1413–1432, DOI 10.1007/s11222-016-9696-4
Weller SC, Vickers B, Bernard HR, Blackburn AM, Borgatti S, Gravlee CC, Johnson JC (2018) Open-ended interview questions and saturation. PLOS ONE 13(6):1–18, DOI 10.1371/journal.pone.0198606, URL https://doi.org/

10.1371/journal.pone.0198606

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

Yli-Huumo J, Maglyas A, Smolander K (2014) The sources and approaches to management of technical debt: a case study of two product lines in a middle-size finnish software company. In: International Conference on Product-Focused Software Process Improvement, Springer, pp 93–107