Towards Grounded Theory Perspectives of Cognitive Load in Software Engineering

Daniel Helgesson
Dept. of Computer Science
Lund University
daniel.helgesson@cs.lth.se

Per Runeson
Dept. of Computer Science
Lund University
per.runeson@cs.lth.se

Abstract

Background: The socio-technical characteristic of software engineering is acknowledged by many, while the technical side still dominates research. As software engineering is a human-intensive activity, the cognitive side of software engineering needs more exploration when trying to improve its efficiency.

Aim: The aim of this study is to increase the understanding of the impact of cognitive load in software engineering. Our ultimate goal is to theorize the knowledge and thereby reveal opportunities to make software engineering more efficient for companies and compelling for the developers to engage in.

Method: We synthesize knowledge, using a grounded theory approach, from our empirical observations and literature on cognitive load in software engineering, using *cognitive load theory* as a stepping stone and theoretical filter.

Results: We present a grounded theory of cognitive load in software engineering, emerging from the analysis, which classifies *cognitive load drivers* into eight *perspectives* – *task*, *environment*, *information*, *tool*, *communication*, *interruption*, *structure* and *temporal* – each of which is further detailed.

Conclusion: We intend to use this theory as a starting point for further generation of theory to be used in the design of software engineering tools, methods and organizational structures to improve efficiency and developer satisfaction by reducing the cognitive load.

1. Introduction

Software engineering is a socio-technical endeavour (Bertelsen, 1997), where the technical side of the phenomena seems to be more studied than the social side (Lenberg, Feldt, & Wallgren, 2015). There has been a thread of research with a social focus (Storey, Ernst, Williams, & Kalliamvakou, 2020), and software engineering is acknowledged as an interdisciplinary field (Méndez Fernández & Passoth, 2019) although the mainstream research still seems to be technology focused, with the social and human aspects considered as the context, at best. Program comprehension research is an exception, integrating cognitive and technical aspects on equal terms (Siegmund, 2016). However, we propose cognitive aspects to be addressed in the broader scope of software engineering as well.

Software engineering always has been (Naur & Randell, 1969), and still is, concerned with efficiency and productivity. However, the constantly growing size and complexity of software systems and development projects, puts increasing pressure on organizations to be efficient in their development and maintenance of software products and services. Research on productivity is typically devoted to specific SE activities, as observed by Duarte's recent literature review (2019). However, the human and cognitive aspects begin to appear more broadly, for example, in Sadowski and Zimmermann's collection on 'Rethinking Productivity in Software Engineering' (2019). Cognitive aspects discussed are primarily related to interruptions in the work environment, although cognitive load and psychological distress are presented as types of *waste* by Sedano, Ralph, and Péraire (2017, 2019). In addition to negatively impacting productivity, these aspects also influence the working conditions for software engineers, impacting on their cognitive sustainability.

We studied *cognitive load drivers* (i.e. *causes* or *sources* of cognitive load) in an industrial case study (Helgesson, Engström, Runeson, & Bjarnason, 2019), rendering an initial taxonomy consisting of *tools*, *information*, and *work & process* type of factors, inducing cognitive load in software engineering. Secondly, we studied cognitive load drivers from novices' point of view, using grounded theory ethnography, pointing us to version control, branching and merge operations adding to the cognitive load of developers (Helgesson, Appelquist, & Runeson, 2021). To advance this line of research, we here survey software engineering literature and use a grounded theory approach to synthesize existing knowledge into a generated theory of cognitive load in software engineering. We use *cognitive load theory* (CLT) by Sweller et al. (1994; 1998) as a stepping stone into the analysis, and also survey its impact in software engineering research. The outcome is a grounded theory, *Perspectives*, which provides abstractions for reasoning on cognitive load drivers in software engineering on which we will base further theory generation and design work for solutions.

The remainder of this paper is structured as follows: Section 2 presents CLT, critique on CLT and use of cognitive load theory in software engineering research. Section 3 presents methodology and describes grounded theory, research goals, data collection, analysis and literature review. Section 4 presents the findings from literature review. Section 5 presents a discussion of the constructs of CLT and synthesis of our findings. Section 6 presents emerging sensitizing concepts and future research, while Section 7 presents a reflection on validity and generalisation issues. Section 8 concludes the paper.

2. Background

It has since Miller's seminal paper 'The magic number seven plus/minus two' (1956) been generally accepted that the human working memory is finite and limited. As the human bandwidth for information processing is limited, it is a trivial argument that reducing unnecessary cognitive load in knowledge based activities (in all likelihood) will free up cognitive resources.

Being a knowledge (and cognitively) intensive activity, cognitive load is a phenomenon that is present in most, if not all, software engineering dimensions (Sedano et al., 2017). In an initial explorative case study (Helgesson et al., 2019) we set out to explore cognitive load from a practitioner point of view, while charting relevant theory from the scientific field of cognition usable for further exploration and description of cognitive load and chart what dimensions of cognition have been have historically been used in the context of software engineering. The study provided us with a first classification of the *cognitive load drivers* (Helgesson et al., 2019) experienced by the practitioners we interviewed. This initial classification of cognitive load drivers gravitated around three main clusters, namely *tools*, *information* and *organisation*.

We noted that program comprehension (Siegmund, 2016) is one dimension of software engineering where cognition has been thoroughly used. Further we noted that cognitive load is a phenomenon that, to some extent, have been explored in other software engineering contexts, but commonly from the perspective of attempting to measure cognitive load (Gonçales, Farias, da Silva, & Fessler, 2019). Further, the exploration of cognition pointed us towards two cognitive directions to further investigate and describe cognitive load, *distributed cognition* (Hutchins, 1995) (Hollan, Hutchins, & Kirsh, 2000) and *cognitive load theory* (Sweller & Chandler, 1994) (Sweller et al., 1998).

Following our first explorative case study, we conducted a larger comparative case study (Helgesson et al., 2021) studying cognitive load drivers *from the novice point of view* by means of grounded theory ethnography (Charmaz, 2014) (Charmaz & Mitchell, 2001), using distributed cognition as theoretical filter. The observed phenomenon we chose to pursue and explain was version control and merge operations which, to our surprise, (systematically) was the largest cause of concern for all of the ten–person teams we observed.

2.1. Cognitive load theory

In *cognitive load theory* (CLT), Sweller et al. have explored cognitive load from a learning perspective (Sweller & Chandler, 1994) (Sweller et al., 1998), when learning complex tasks. Paas, Renkl, and Sweller (2004) state that: "It is generally accepted that performance degrades at the cognitive load extremes of either excessively low load (under-load) or excessively high load (overload)" and that "learning situations with an extremely high load will benefit from practice conditions that reduce load to more manageable levels".

In CLT cognitive load components are classified as *intrinsic* (i.e. related to the cognitive task), *extraneous* (i.e. the way information is presented to the subject) or *germane* (i.e. the cognitive cost of learning from the observation/problem solving) (Sweller & Chandler, 1994) (Hollender, Hofmann, Deneke, & Schmitz, 2010) (Sedano et al., 2017). CLT is based on the model of working memory provided by Baddeley (1976, 1992), assuming that information processing and storing are two separate, yet interdependent processes. A central tenet in CLT is the assumption, analogously to Miller (1956), that the capacity of the human working memory is not only finite but also limited and further, that information processing, storage and retrieval will use parts of these finite resource (Debue & van de Leemput, 2014).

While CLT serves as a stepping stone in the process of understanding the fundamental traits of cognitive load, it should be noted that it developed in a *learning* context rather than *problem solving-*, or *task solving* perspectives in general. A SLR (Snyder, 2019) (n=65) describing the similarities and differences between CLT and Human Computer Interaction (HCI) was executed by Hollender et al. (2010), and it provides a thorough description and explanation of the fundamental phenomena associated to CLT (e.g. *germane load*, *intrinsic load*, *extraneous load*, *worked example effect*, *split-attention effect*, *modality effect* and *redundancy effect*) as well as a comparison between CLT and HCI.

2.2. Critique on CLT

Hollender et al. (2010) further highlight relevant criticism of CLT, namely that the principal components of cognitive load *intrinsic*, *extraneous* and *germane* cognitive load to some extent "belong to different ontologies" and simply are not additive (intrinsic load refers to the complexity of the task and germane load to the process of creating knowledge per se) as pointed out by de Jong (2010). The critique has been elaborated by Moreno (2010). Debue and van de Leemput (2014) provide a further discussion on the critique of cognitive load theory.

From our vantage point the critique by de Jong and Moreno, respectively, appears quite valid. Germane load differs considerably from task intrinsic and extraneous load (it does however present an important perspective on cognitive load in software engineering). Further, the different principal components of cognitive load are possibly (or probably) not *additive* – but we argue that they are closely enough related and that they arguably drain cognitive resources regardless of their relation.

This is the very reason why we propose our theory of cognitive load in software engineering. We argue that our current direction, and key assumption, is valid, since it is possible to identify cognitive load drivers regardless of their nature in terms of being *additive*, they are for all points and purposes *communicating vessels* drawing from the same finite and limited resource (the human working mind/memory).

2.3. Use of CLT in Software Engineering Research

Cognitive load, as phenomenon, has to some extent been studied in software engineering contexts. Gonçales et al. (2019) list 33 studies in a systematic mapping study and conclude that the phenomenon appears mostly to be studied from a measurement perspective, i.e., measuring cognitive load using eye-tracking and other biometric/psychophysiological sensors (e.g. Fritz and Müller (2016)). Program comprehension is one dimension of software engineering where cognitive load has been studied (Siegmund, 2016).

In their paper on *software development waste*, Sedano et al. (2017) describe a constructivist grounded theory study at a case company over two years, resulting in a taxonomy of software waste. While not specifically looking at cognitive load, one of the entities in the resulting taxonomy is derived from CLT, namely *extraneous cognitive load*, consisting of *overcomplicated stories*, *ineffective tooling*, *technical debt* and *multitasking*. Further they added *psychological distress*, *waiting/multitasking* and *ineffective communication* as entities in their taxonomy of waste.

Section 4 presents the findings from our literature review on use of CLT in Software Engineering, indicating that the impact of CLT in SE has thus far been small.

3. Method

3.1. Grounded theory

In this study we conduct a qualitative synthesis of two grounded theory studies and extant literature towards theory generation. While the use of qualitative synthesis is not mainstream in the Software Engineering research community, the use of qualitative syntheisis in SE has been studied by Cruzes and Dybå (2011b) on research synthesis, and the same authors have since published guidelines for synthetical research activities (Cruzes & Dybå, 2011a). The authors list 13 different methods including grounded theory (Stol, Ralph, & Fitzgerald, 2016).

Grounded theory (GT) (Glaser & Strauss, 1967) originated in the mid sixties as a qualitative reaction to the positivistic and quantitative research paradigms at the time dominant in the social sciences (Charmaz, 2014). It has since gained considerable traction in many fields outside of social sciences, and diverged into three main currents (Charmaz, 2014) (Stol et al., 2016).

Inherently inductive (Charmaz, 2014), or abductive (Martin, 2019), and iterative (Charmaz, 2014), the main purpose of GT is to allow for generation of 'theory'. While we have not yet reached a fully mature, substansive, theory we still see our contribution as more than a conceptual model, i.e. a *theory* (Abend, 2008).

One aspect of grounded theory that has been thoroughly discussed is that of the role of literature. It its original form the the GT manifesto (Glaser & Strauss, 1967) has been interpreted as no literature should be consulted prior to the analysis (Charmaz, 2014) (Martin, 2019). Yet Glaser himself states that: "...reading and use of literature is not forsaken in the beginning of a grounded theory project. It is vital to be reading and studying from the outset of the research, *but in unrelated fields*" (Glaser, 1992, p. 35).

In the last few years numerous texts on the use of literature and abductive reasoning in the grounded theory studies has been published (e.g. (Thornberg & Dunne, 2019) (Martin, 2019) (Bryant & Charmaz, 2019) (Tavory & Timmermans, 2019) (Gorra, 2019)) indicating that while there is a will within the grounded theory research community to further extend the role of literature in grounded theory research. Reflecting on the above and the original GT dictum stating that "all is data" (Stol et al., 2016) (and that "the word *data* seems to mean whatever Glaser or Strauss arbitrarily choose it to mean" (Alvesson & Sköldberg, 2018)) we do not really see an issue with fusing empirical data with literature. Further (Dey, 1993) stresses: "In short, there is a difference between an open mind and an empty head. To analyse data we need to use accumulated data, not dispense with it. The issue is not whether to use existing knowledge, but how."

We take a pragmatic postpositivist (Robson, 2002) epistemological position in this paper. Our aim is to provide a grounded theory for reasoning on cognitive load in software engineering, using abductive reasoning on literature and data, and our ambition is to provide knowledge for software engineering research community and practitioners. We use grounded theory as a method, not an epistemological position – our take is that the epistemological position should reflect the nature of phenomenon under study, not the other way around.

3.2. Research goals

Central to Glasers as well as Charmaz grounded theory versions is that the final research questions are not defined up front at the beginning of the research project. In the first case, Glaser suggests that the researcher should start with *area of interest* (Glaser, 1992) (Stol et al., 2016), while Charmaz suggests that the researcher should start with *initial research questions* that evolve through the study (Charmaz, 2014) (Stol et al., 2016).

In this study our research goal is to fuse the observations made in two previous case studies by means of abductive reasoning on data and extant literature. Our research goal is twofold:

A) To generate and present a grounded theory for reasoning on cognitive load in a software engineering context from abductive synthesis of empirical data and literature using cognitive load theory as a theoretical lens; and secondly

B) To explore cognitive load theory, and its use in the software engineering research community.

3.3. Data collection

Study I (SI) – In the first case study (Helgesson et al., 2019), we set out to document the experience and consequence of cognitive load of professional software developers at a large (1000+ developers), international software development organisation in the telecom and mobile device sector. The study consisted of 5 semistructured interviews, that was analysed using thematic analysis (Braun & Clarke, 2006).

The design of this study was not formulated as GT, but as an *explorative case study* (Runeson, Höst, Rainer, & Regnell, 2012). While not positioned as GT it did contain a considerable degree of GT practices: it was *explorative*, it used *initial research goals* rather than preformulated research questions, it featured *iterative data collection and analysis*, as well as *open coding* and we returned to the field for additional data after the first round of analysis in order to achieve some *triangulation*.

The result of the first study was an initial classification of the *cognitive load drivers* we encountered during the analysis. From a grounded theory perspective this initial classification would, largely, correspond to *sensitizing concepts* (Charmaz, 2014).

Study II (SII) – In the second case study (Helgesson et al., 2021), we set out to chart *cognitive load drivers* from a *novice point of view* using grounded theory ethnography (Charmaz & Mitchell, 2001) (Charmaz, 2014). The case we investigated by observation, was four different ten–person teams of sophomore computer engineering students, working together for a semester (one full day each week) as an agile development team developing a software system. The data we collected consisted of weekly individual reports written by the students, a weekly questionnaire to each student, field notes, focus groups, field experiments and three short follow up interviews to allow for an increase in saturation following the first round of open coding.

This study was specifically designed as grounded theory ethnography, and largely adhered to the guidelines provided by Charmaz (Charmaz, 2014), in terms of data collection and analysis. The study used Distributed Cognition as theoretical lens, compared four different teams and we went through stages of *open*, *focused* and *theoretical coding* (Charmaz, 2014), using *memos/memoing* (Glaser, 1978) as main means of analysis.

The result of the second study was a theoretical explanation of the largest cognitive load driver, or phenomenon, that we observed; namely that version control and merge operations was the largest cause of concern for all the teams studied. However, the rich data we collected contained more observations and findings than what we could use in the following theory generation.

3.4. Analysis

This paper originated as a memo describing *sensitizing concepts* (Charmaz, 2014) in regards to cognitive load in software engineering and CLT. Open, focused and theoretical coding of the result of the previous two studies (SI, SII) and the findings of Sedano et al. (2017) was done by the first author alternating between post-it stickers on whiteboards and memos.

Following the thoughts on abductive reasoning, presented by Martin (2019) the findings were processed in *memo* form. It could be described as *memo*-iterative and *memo*-exploratory, firstly using *open* and later *focused* and *theoretical* coding.

4. Literature review on Cognitive Load Theory in a general Software Engineering context

In order to chart the impact of cognitive load theory in software engineering research community we performed a limited literature study.

We based our literature search strategy on that of Hollender et al. (2010), as used to chart CLT in HCI. They limited their search to querying ACM only and let the results serve as a proxy, rather than completing an exhaustive search. We queried the ACM Fulltext library and IEEE respectively with the queries:

ACM `"Cognitive Load Theory" AND Sweller AND (Software AND (Development OR Engineering))`

IEEE `"Full Text & Metadata":"cognitive load theory" AND ("Full Text & Metadata":"Software Development" OR "Full Text & Metadata":"Software Engineering")`

We had to omit ‘Sweller’ from the search string for IEEE on account of issues with indexing of reference section (e.g. Sedano et al. (2017) does not show up when ’Sweller’ is included as part of the query, as it only references Sweller, the name is not mentioned in the actual article), and later filter out and remove papers that contain no reference of Sweller.

We decided to use the past five years (i.e. 2015–2020) as a timeframe. Book sections excluded, we found 22 papers in ACM and another 65 papers in IEEE matching the queries. We further manually excluded papers regarding *teaching* or *education*, short papers, posters and papers related to general HCI, and ended up with an aggregated result of 11 relevant papers. The first author initially defined the first set of papers for exclusion, and marginal papers were reexamined by the first and second author collectively.

We only found one relevant study of cognitive load in a general software engineering context using cognitive load theory, namely Sedano et al. who conducted a long term grounded theory study at a case company (Sedano et al., 2017), resulting in a taxonomy of software waste. One of the clusters found was *extraneous cognitive load*. Their findings are in line with what we have found in previous studies (SI), (SII).

Further, Krancher and Dibbern (2015) present a multiple case study investigating the importance of knowledge in software maintenance outsourcing.

4.1. Measuring cognitive load

The largest cluster of papers we found gravitated around *measurements of cognitive load*.

Gonçales et al. (2019) conducted a systematic mapping study specifically addressing measuring the cognitive load of software developers. The authors found 33 papers, and provide a classification of the articles found. We note a certain overlap with the papers we found, e.g. Müller and Fritz (2016) and Crk, Kluthe, and Stefik (2016). This mapping study has since we executed the literature review been extensively extended (Gonçales, Farias, & da Silva, 2021). Fritz & Müller present two papers; on the use of sensor driven ’biometrics’ to boost software developer productivity (Fritz & Müller, 2016) and a case study aimed at predicting code quality online using various sensors (Müller & Fritz, 2016). Karras, Risch, and Schneider (2018) used eye-tracking to study the impact of different linking variants of use cases and associated requirements on reading behaviour. Crk et al. (2016) present an empirical study in which programming expertise is explored using brain wave changes (EEG).

4.2. Improving software development

We found two papers related to improvement of software engineering activities. Henley and Fleming (2016) present at tool for improving code change support in visual dataflow programming environments while Moseler, Wolz, and Diehl (2020) present a prototype tool for visualising debugging scenarios.

Table 1 – Cognitive load perspectives grouped by cognitive load theory components and association to data set

CLT component	**Perspective**	**SI**	**SII**	**Sedano**
Intrinsic cognitive load	Task	x	x	x
Germane cognitive load	Environmental	x	x	x
Extraneous cognitive load	Information	x	x	x
	Tool	x	x	x
	Communication	x	x	x
	Structural	x	x	x
	Interruption	x	x	x
	Temporal	x	x	

4.3. Bordering on software engineering
Bordering on software engineering, Kelleher and Hnin (2019) describe an approach to predict the cognitive load of code puzzles. In addition we found a proposed model for API learning by Kelleher and Ichinco (2019), and an explorative analysis of the notational characteristics of decision models by Dangarska, Figl, and Mendling (2016).

4.4. Summary of literature review
In summary we find the cognitive load, as such, is a known phenomenon in software engineering and that it has been found to be explored and evaluated using metrics and sensors. We further find that CLT has not had a thorough impact on the software engineering community, but that Sedano et al. (2017) specifically use CLT and *extraneous cognitive load* in their classification of *software waste*.

5. Perspectives – Result
In this section we discuss the constructs of CLT and synthesise our findings from our previous work (SI, SII) and that of Sedano et al. (2017) in order to generate a grounded theory for reasoning on cognitive load in the work environment of software engineers.

We present eight (8) different *Perspectives* (i.e. Task, Environmental, Structural, Information, Tool, Communication, Interruption and Temporal) on cognitive load in software engineering. These *perspectives* consist of categories of *cognitive load drivers* (Helgesson et al., 2019) and our intention is to raise the abstraction level when reasoning on cognitive load and cognitive load drivers. See Table 1 for an overview of the *perspectives* and how they are derived from field studies and their relation to CLT components.

We refer to them as perspectives, since they are not distinct aspects of cognitive load, but rather a set of lenses through which we observe and analyse the phenomenon of cognitive load in software engineering activities. From the analysis, six implications for design of software engineering tools and practice, emerge as sensitizing concepts (Charmaz, 2014) (marked [SC1–6]). They are further described in Section 6.

5.1. Reflection on cognitive load and cognitive load theory
As stated in Section 2, in CLT three different components of cognitive load are suggested – *Intrinsic*, *Extraneous* and *Germane*. The *intrinsic* load is defined as the load of the cognitive task to be solved, the *extraneous* load as the cognitive load resulting from task presentation or environment, and the *germane* load refers to the cognitive resources used for learning or internalising schemas for problem solving (Debue & van de Leemput, 2014) (Hollender et al., 2010). In their critique on CLT, de Jong (2010) and Moreno (2010), respectively point out the principal components of CLT belong to “different ontologies”. We would like to point out that in our observations the nature of cognitive load is often overlapping, depending on what *perspective* the observer choses as a lens [SC1].

5.2. Intrinsic cognitive load

Sedano et al. (2017) aptly highlight that many (if not most) software development activities are *cognitively intensive*, i.e. that these activities consist of tasks that have a relatively high intrinsic cognitive task load. They suggest *overcomplicated stories* as one of their identified sources, or drivers, of *extraneous cognitive load*. We suggest analysing an overcomplicated story serves as one example of a task that should be reduced in complexity in order to reduce the cognitive load of the individual developer.

In our empirical data we have in several instances observed that the cognitive task it self (or, the actual *design of the task* can be a cognitive load driver of considerable magnitude. For instance in the absence of automation we see engineers performing tedious manual tasks that, ideally, should be automated (SI). We also noted that users had to fill out very intricate and detailed sheets of information when reporting issues, supplying information that was no longer used by anyone (SI).

Further we observe that if a *task* is closely associated to the use of a tool, the distinction between the task intrinsic cognitive load and the extraneous, *external*, cognitive load induced on the user by the tool becomes difficult, if not impossible, to pinpoint (SII). From a software engineering perspective, which refers more to *solving problems* and *completing tasks* rather than *learning* per se, it seems that the design of the task it self is an essential perspective of cognitive load.

The rationale for allowing *task* as a *cognitive load driver*, or *perspective* is that tasks themselves can induce cognitive load if they are designed wrong, overly complex or if they depend on engineers spending mental effort on tedious chores that could/should be automated [SC2].

We thus continue our reflection on CLT by suggesting a:

5.2.1. Task perspective

A task centric perspective of cognitive load in software engineering is warranted by the *cognitively intensive* nature of software engineering. We conlude that *task/-s too complex* as observed by Sedano et al. (2017) present one important aspect of cognitive load in software engineering. Further we note that some tool related tasks need additional user support (SI, SII), and claim that the higher the cognitive load in the task the higher the reward in easening of the cognitive load situation of the engineer. We conclude by observing that *task needing automatization* (SI) and *unnecessary tasks – waste* (SI), e.g. filling out unnecessary forms or manually moving data from one tool to another, present one dimension of cognitive load that we consider a distinct *waste* in software engineering.

5.3. Germane cognitive load

As described in Section 2 the *germane* cognitive load in CLT refers to the cognitive resources and processes devoted to acquisition, retention and automation of schemata for the task at hand (Debue & van de Leemput, 2014) – i.e. the cognitive cost of learning, and the distinction between intrinsic and germane load is debated. While not drilling too far into the ontological issues of cognitive load theory, we note that the discussions on the matters presents an additional possible perspective of cognitive load drivers in software engineering – namely a general cognitive perspective. We know from seminal schemata theory that novices and experts often display vastly different strategies while solving identical problems (Chi, Glaser, & Rees, 1981) (Chi, Glaser, & Farr, 2014). Similarly novices and expert have different needs in terms of cognitive support in digital tools (Moody, 2009) (Vessey, 1991) (Vessey & Galletta, 1991), and we have analogously observed that novices have different needs in terms of support from the software development environment compared to experienced developers (SII).

Further, we can observe that *learning* something does have a cognitive cost. While this can be trivially observed, it has an interesting consequence. If the internalisation of a problem (or task) solving schema comes at a cognitive cost, what happens when something has to be *relearned* (e.g. when a new tool, IDE, operating system or programming language is introduced). Consider a schemata for solving a specific task or problem that has been internalised to the point of automation. What happens when a new similar, yet different, schemata must internalised? This activity in all likelihood comes with a cognitive cost. Further, in the situation where several competing schemata have been internalised this will in all likelihood be even worse.

In our first case study (SI), we noted that one of the interview subjects described the migration from one issue management system to another as troublesome, on account of the new system not matching his mental model of how the system operated. In our second case study (SII) we found that a large portion of the subject described the complexity and intricacies of GIT as a cause of negative stress. Analogously Sedano et al. (2017) noted noted *psychological distress* as a specific type of *software waste*.

Further, in (SII) we noted that several students described their experience of working for an entire day in a computer lab as *draining* on account of various reasons. We noted students stating themselves as being *introvert* and found being in close proximity of other people (e.g. through pair programming) as *very draining*, and we further noted students stating that they did not understand how it would be possible to work under these conditions for a normal 40 hour work week, on account of noise, light, interruptions and lack of oxygen.

These sources of cognitive load in a work setting are in line with the findings of a case study by Sykes (2011) on interruptions in the work place. The author highlights the importance of the physical workplace environment and sound levels in the working areas. Further Sykes note that the while the use of headphones augments blocking out office noises, it is essentially a "band-aid solution to the root problem". Further, Kirsh (2000) has described the consequences of *cognitive overload* in a general workplace setting [SC3].

We do not really see that these dimensions of cognitive load as possible to map to either of the two main constructs of CLT, *intrinsic* and *extraneous*. None the less, we find them too important not to mention in this context. As a result we propose an *environmental* perspective of *cognitive load drivers* in Software Engineering, analogous to *germane* cognitive load.

5.3.1. Environmental perspective

An environmental centric perspective of cognitive load in software engineering consists of cognitive ergonomic factors, ergonomic factors and psychological factors. While these constructs, to some extent are overlapping, we still note them as highly relevant when analysing the cognitive work environment and the cognitive load situation of software engineers.

5.4. Extraneous cognitive load

In CLT the component of *extraneous load* refers to the cognitive load resulting from *task presentation* or *environment* (Sedano et al., 2017). Given that the human bandwidth for cognitive load is limited (Miller, 1956), an increase in extraneous cognitive load will reduce the amount of cognitive bandwidth available for task solving (intrinsic task load) and for the cognitive processes of learning (or problem solving).

In their report on a grounded theory case study on *software development waste* Sedano et al. (2017) state that: a) since many software development activities have a high intrinsic cognitive load, and b) the mental capacity of the individual developer is a limited resource, and that they, as a consequence, see *extraneous cognitive load* as *waste*. They further use *extraneous cognitive load* as a catch all element in their waste taxonomy, containing *technical debt*, *inefficient tooling*, *waiting/multitasking*, *inefficient development flow* and *poorly organised code*. Further, Sedano et al. also identify *inefficient communication* and *psychological distress* as two different types of waste in software development outside of *Extraneous Cognitive Load*.

We are in complete agreement on the importance of reducing extraneous cognitive load on the individual developer, and that developers spending mental effort on managing *inefficient tools* is definitely to be considered waste. In our first explorative study (SI) of cognitive load in software engineering we noted a *structural* perspective of cognitive load drivers associated to *work, process & organisation*. Further, we found cognitive load drivers clustered around *information* and *tools*.

5.4.1. Structural perspective

A structurally oriented perspective on cognitive load in software engineering, as we see it, consists of organisational legacy, structure and processes. Sedano et al. (2017) present *technical debt*, *poorly organised code* and *inefficient development flow* as examples of *extraneous cognitive load*. We have observed similar findings from a developer point of view in large software organisation (SI, SII): *ad hoc implementation of process*, *ad hoc implementation of information structure*, *ad hoc implementation of tooling* and *lack of understanding of organisation*.

5.4.2. Information perspective

An information centric perspective on cognitive load in software engineering reflects on the nature of Information and its consequences for the individual developer. Sedano et al. (2017) present *ineffective communication* as one specific type of *software development waste*, but we choose to distinguish between *information* and *communication* in our theory – in an *information centric* perspective the phenomena under study are associated to the nature of the construct 'information'; in a *communication centric* perspective the phenomena under study are associated to 'communication and distribution' of information. In our first study (SI) we noted two different aspects of information relevant, *integrity of information* (i.e. the reliability and completeness of information) and the *organisation of information* (where to find information and knowing where to distribute information to). We have also noted that the way information is structured and presented can be a cause of cognitive load (overview and details). The observations in (SI) were largely validated by observation in SII.

We note that software engineering activities are information centric. The revolve around *information* that can be classified into two groups: *essential information* and *meta information*. The former equates *source code*, the latter *information about source codes* (e.g. bug reports, requirements, specifications, use cases etc.). This can aslo be viewed as *essential instructions* (i.e. *source code*) and *meta instructions* (i.e. *instructions* on how to *create/transform/synthezise source code*) [SC4].

5.4.3. Tool perspective

Since most, if not all, software development relies on tools and toolchains we consider a tool centric perspective of Cognitive Load in software engineering as being merited. Outside of IDEs, source code editors and compilers, developers also use version control tools, merge tools, test tools etc.. This is fully inline with our observations in SII and as a consequence, we find the tool centric perspective quite important.

Sedano et al. (2017) simply state that *inefficient tools and problematic APIs, libraries and frameworks* are an observed cause of cognitive load. We have observed (SI, SII) that cognitive load is induced on the developers by tools in several different ways. Lack of needed functionality forces the developer to waste effort when forced to manually do something that the tool does not support, or as we noted in our first study where missing search functionality prevented users to find older, closed, defect reports in an issue management system. In that specific case our informant saved all notification emails from the issue management system, and used that as his searchable system, using the email client. This overlap with the *temporal perspective* serves as an example on how different the drivers of cognitive load can appear depending on what perspective one takes.

We also noted that the stability, and reliability, of tools were important in terms of cognitive load. Being able to revert user errors (e.g. Git) is important and so is understanding and trusting the result of an automatic merge operation (SII). Further, we have noted that developers get frustrated when a tool crashes and all work is lost (SI). Our main example draws on an issue management system in which the developers were forced to fill out several forms that were quite complex, required considerable amounts of irrelevant data and were somewhat unstable – leading the developers to lose all the data that they had entered and forcing them to redo the entire operation. Stability and reliability also includes *downtime*, that is a system that is unavailable; as well as *lag* where the tool freezes up momentarily resulting in a loss of focus on the user.

We further have noted interaction issues in relation to tools as a significant contributor to cognitive load for software development. In our first study (SI) we noted that *unintuitive and cumbersome interaction* of a tool and *lack of integration* of tools was a concern for developers. We further noted that inconsistencies between different aspects of a tool or between two different tools were considerable load drivers. These findings were largely validated by our second study (SII).

In conclusion, a tool centric perspective on cognitive load in software engineering gravitates around tools lacking functionality; the fitness to purpose of the tool as well as unintuitive, cumbersome and inconsistent user interaction. It further includes *lack of integration* between different tools and involves the *reliability* as well as *stability* of the tools.

5.4.4. Communication perspective

A communication centric perspective of cognitive load in software engineering is derived equally from (SII) and the findings of Sedano et al. The phenomena under study are associated to the *process of distributing information* rather than to the *nature of information itself.* Sedano et al. (2017) propose *inefficient communication* as one specific type of waste in software development, describing it as 'the cost of incomplete, incorrect, misleading, inefficient or absent communication'.

While we did not specifically explore *communication* as a cognitive load driver in our first study (SI), there were appearances of phenomena related to communication. We noted these issues as aspects as related to *information distribution*. Our second study (SII) largely validated *communication* as a significant cause of cognitive load of the developers.

We noted issues on the individual level, in knowing whom to communicate with, from an *information retrieval perspective* (i.e. *whom to ask/where to look*) as well as from an *information distribution perspective* (i.e. *whom to inform/where to store*). We further observed issues on group level analogous to the observations, e.g. absent communication leading to team members misunderstanding each other or actual waste when multiple of developers are working on the same issue without knowing about it because of absent stand/up meetings, or simply in inefficient meetings.

5.4.5. Interruption (& multitasking) perspective

An interruption (and multitasking) centric perspective of cognitive load in software engineering is merited on account of the social nature of the endavour. Interruptions and multitasking is an integral part of modern software engineering (Brumby, Janssen, & Mark, 2019). Sedano et al. (2017) noted *unnecessary context switching* as one aspect of *extraneous cognitive load*, while presenting *waiting/multitasking* as another type of *waste* in software engineering, outside of *extraneous cognitive load*. We noted indications of *interruptions* as a cognitive load driver in our first study (SI), mostly attributed to tool stability. In our second study (SII) we noted interruptions as a consequence of the developers shifting pairs, and describing effect of the task switching as a *loss of flow*.

Sykes (2011) reports on interruptions in software engineering in a case study, presenting findings that indicate interruptions to be a significant cognitive load driver: "Aggregated data extrapolated over a typical 8-h work day translates into over 120 interruptions per day for Technical Lead/Senior Developers and accounts for 5.7 h of time working on interruption tasks. This translates into over 71% of their daily activity is spent on dealing with interruptions".

Further, Sykes highlights that "there is a strong correlation between cognitive load and the cost of interruption", i.e. interruption of a task with high intrinsic cognitive task load will result in a longer *resumption lag*. The obvious consequence of this is that people performing high cognitive activities, such as software engineering/development tasks are likely to be significantly impacted by interruptions and the overall productivity will decrease on account on the longer resumption lags. It is also highlighted that interruptions drive stress, or "negative emotions, such as, irritation, or frustration".

5.4.6. Temporal perspective

A time centric, or *temporal*, perspective of cognitive load in software engineering has, thus far, proven quite elusive. While becoming a somewhat more tangible concept throughout the iterations of research cycles a precise definition of *what* a temporal perspective of cognitive load is remains elusive [SC5]. However, revisiting the material from our first two studies we note a specific temporal perspective of cognitive load. In our initial field study we noted *temporal traceability* as one aspect of cognitive load associated to the Information cluster. We noted developers having issues with version control and trouble finding closed error reports (i.e. events occurring in the past). Specifically, in one case the developer utilised a folder in the email client to create a separate and searchable record of closed issues. Again we note the overlap with previous perspectives.

In our second study we noted that version control and merge operations were the main source of cognitive load in the four development teams we studied. Further we noted that the fundamental temporal aspect of distributed cognition (Hollan et al., 2000), that cognitive processes can be distributed in time so that earlier events "can transform the nature of later events" was clearly visible in the observations. We further observed a number of cognitive load drivers associated to the temporal perspective, primarily we observed the complexities presented by configuration management tools and branching strategies. When looking at the reflection of Hollan et al. on *history enriched objects* it is hard not to see parallells in version control and merge operations.

We also note that while most tasks in software engineering requires bridging of a *cognitive gap* (e.g. the transformation of a requirement to a specification, or the transformation of a specification to source code), the *synthesis* of a merge operation specifically bridges a *temporal gap* in the production of software, in the sense that the components fo the merge operations (*meta information* and *essential information*) was produced at an earlier stage, possibly by someone else [SC6].

We further noted a temporal aspect of understanding project situations in our observations. This is not only about a momentary snapshot, but about cognitive processes distributed over time. The question of project overview in a distributed agile project quickly becomes multidimensional, seeking to answer *who did what, when, where and why?* – not unlike the concept of *ba*[1], the heideggerian "space±time nexus" described by Nonaka et al. (2000).

6. Sensitizing Concepts & Future Research

While working on memos for this manuscript we noted some *sensitizing concepts* (Charmaz, 2014) from a design science perspective. Space limitations does not allow for an in depth reasoning, so they are shortly described as senzitising concepts 1–6 below:

1. We observe that the concept of *cognitive productivity* represents the software development organization's strive for productivity and efficiency, while *cognitive sustainability* addresses the developers' wellbeing, which indirectly, of course, also affects the development organization. We wish however not only to explore *cognitive productivity*, *efficiency* and *amplification*, we also want to bring attention to *cognitive sustainability*.

2. We note that higher the intrinsic task load of a tool supported task, the more investment in user support and training on the tool can be motivated.

3. Our observations of cognitive load induced by relearning leads to 1) considering design of configurable tools to enable personal adaptation, and 2) questioning too frequent upgrading pace of tools to reduce relearning load.

4. From the information centric perspective we note two kinds of information in software engineering – *essential information* and *meta information*, while further observing a duality in the nature of *information* – it can be described as either *information* or *instructions*.

[1]"Ba does not necessarily mean a physical space. The Japanese word 'ba' means not just a physical space, but a speciic time and space. Ba is a time±space nexus, or as Heidegger expressed it, a locationality that simultaneously includes space and time. It is a concept that unifies physical space such as an office space, virtual space such as e-mail, and mental space such as shared ideals."

5. While we have only looked at the temporal perspective as the relation between the past and the present thus far, it may also well extend into understanding, and predicting, the future, activities that in all likelihood induce considerable cognitive load.

6. We noted that merge operations, a temporal synthesis of meta information (e.g. commit messages) and multiple sources of essential information (i.e. two different versions of source code), seem to to be harder than actual coding (production of essential information). Essentially we note that the a task consisting of synthesis of essential information and meta information appears to have a higher intrinsic cognitive task load than production of either meta- or essential information, provided that the level of abstraction is comparable. To us this is an indication that, while the additional cognitive user support should always be considered when designing software development tools, it should definitely be further investigated specifically in relation to configuration management, branching and merging.

Further exploration of these concepts will be part of future research in this research project, that will also include:

- an in depth study of literature on cognitive perspective in regards to version control and merge tools
- a study focused on benchmarking existing git integrations in a few a the existing IDEs
- design recommendations based on the consequences of cognitive load in software engineering
- in depth industrial case studies with the aim to further elicit cognitive load drivers in the industry
- further theory building towards a theoretical understanding of cognitive load as a software engineering phenomenon

7. Validity & Generalisation issues

In this paper we present qualitative perspectives of cognitive load drivers in software engineering. These perspectives are grounded in observations and literature. There are in all likelihood other factors affecting the cognitive ergonomic situation of software engineers, but we focus on those we have observed.

GT studies are commonly evaluated based on the following criteria (Sedano et al., 2017) (Charmaz, 2014) (Stol et al., 2016):

Credibility: *Is there enough data to merit claims of perspectives?* This study relies on the data set from two case studies and meta analysis of a third case study. The data set includes interviews, focus groups, observations, written reflections and extant literature

Originality: *Do the perspectives offer new insight?* While cognitive load is not an unknown phenomenon in software engineering, this is one of the first studies that utilise cognitive load theory for analysis of cognitive load in software engineering. Consequently the result can offer novel observations in regards to how to reason on cognitive load in software engineering.

Usefulness: *Are the perspectives relevant for practitioners?* This study identifies novel perspectives on cognitive load in software engineering. If viewed as *waste* (Sedano et al., 2017), reduction of cognitive load in software engineering can be seen as means to increase efficiency and productivity. If viewed as *cognitive work environment issues* (Gulliksen, Lantz, Walldius, Sandblad, & Åborg, 2015), their reduction would equal improving the cognitive sustainability and developer experience in software engineering.

Resonance: *Do the perspectives make sense to participants?* This study identifies novel perspectives on cognitive load in software engineering and they are peer reviewed in this paper. Further validation is planned for in future work.

In regards to our literature study – it is limited, and serves the purpose of acting as an indicator on the magnitude of use of cognitive load theory in software engineering research.

In regards to external validity – grounded theory is a largely qualitative methodology, the findings are not statistical, and can not be statically generalised. That being said, in study I we compared our findings to a general taxonomy of cognitive ergonomics (Gulliksen et al., 2015); in this study we compare our findings to those of Sedano et al. (2017) and we do not see our findings as very particular to the contexts in which we have observed them.

8. Conclusion

As a response to the need for exploration of the social side of software engineering, we derived eight cognitive load perspectives, based on grounded theory analysis of empirical observations of our own in industry (SI), in complex novice settings (SII), and related literature (Sedano et al., 2017). The perspectives are partially overlapping, but constitute still unique view on the cognitive load created in software engineering. Further, we conclude that cognitive load is a known phenomenon in software engineering literature, while cognitive load theory does not appear to have had any major impact.

Acknowledgement

The authors wish to thank E-building librarian, and general go-to-nice-guy, Andreas Karman for helping out with queries and general debugging of ACM and IEEE search results and philosopher-gone-translator Ylva Stålmarck for proofing the manuscript. We further wish to thank the PPIG reviewers for their thoughtful suggestion on improving the paper as well as the PPIG audience for encouraging and insightful discussions. The work described in this paper was conducted in the ELLIIT[2] strategic research environment.

9. References

Abend, G. (2008, June). The Meaning of 'Theory'. *Sociological Theory*, *26*(2), 173–199. (Publisher: SAGE Publications Inc) doi: 10.1111/j.1467-9558.2008.00324.x

Alvesson, M., & Sköldberg, K. (2018). *Reflexive Methodology - New Vistas for Qualitative Research* (3rd ed.). London, UK: SAGE Publications.

Baddeley, A. D. (1976). *The Psychology of Memory*. New York, NY, USA: Basic Books.

Baddeley, A. D. (1992, July). Working Memory: The Interface between Memory and Cognition. *Journal of Cognitive Neuroscience*, *4*(3), 281–288. (Publisher: MIT Press)

Bertelsen, O. W. (1997, November). Toward A Unified Field Of SE Research And Practice. *IEEE Software*, *14*(6), 87–88.

Braun, V., & Clarke, V. (2006, January). Using thematic analysis in psychology. *Qualitative Research in Psychology*, *3*(2), 77–101.

Brumby, D. P., Janssen, C. P., & Mark, G. (2019). How do interruptions affect productivity? In C. Sadowski & T. Zimmermann (Eds.), *Rethinking productivity in software engineering* (pp. 85–107). Berkeley, CA: Apress. doi: 10.1007/978-1-4842-4221-6_9

Bryant, A., & Charmaz, K. (2019). Abduction: The Logic of Discovery of Grounded Theory - An Updated Review. In *The SAGE Handbook of Current Developments in Grounded Theory*. London, UK: SAGE Publications.

Charmaz, K. (2014). *Constructing Grounded Theory* (2nd ed.). London, UK: SAGE Publications.

Charmaz, K., & Mitchell, R. (2001). Grounded Theory in Ethnography. In *Handbook of Ethnography*. London, UK: SAGE Publications.

Chi, M. T. H., Glaser, R., & Farr, M. J. (2014). *The Nature of Expertise*. Psychology Press. doi: 10.4324/9781315799681

Chi, M. T. H., Glaser, R., & Rees, E. (1981, May). *Expertise in Problem Solving*. (Tech. Rep. No. TR-5). Pittsburg Univ PA Learning Research and Development Center.

[2]https://liu.se/elliit

Crk, I., Kluthe, T., & Stefik, A. (2016, February). Understanding Programming Expertise: An Empirical Study of Phasic Brain Wave Changes. *ACM Transactions on Computer-Human Interaction*, *23*(1), 1–29.

Cruzes, D. S., & Dybå, T. (2011a, September). Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement* (pp. 275–284). (ISSN: 1949-3789)

Cruzes, D. S., & Dybå, T. (2011b, May). Research synthesis in software engineering: A tertiary study. *Information and Software Technology*, *53*(5), 440–455.

Dangarska, Z., Figl, K., & Mendling, J. (2016, September). An Explorative Analysis of the Notational Characteristics of the Decision Model and Notation (DMN). In *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)* (pp. 1–9). (ISSN: 2325-6605)

Debue, N., & van de Leemput, C. (2014). What does germane load mean? An empirical contribution to the cognitive load theory. *Frontiers in Psychology*, *5*.

de Jong, T. (2010, March). Cognitive load theory, educational research, and instructional design: some food for thought. *Instructional Science*, *38*(2), 105–134.

Dey, I. (1993). *Qualitative Data Analysis: A user-friendly guide for social scientists*. London, UK: Routledge.

Duarte, C. H. C. (2019, May). The Quest for Productivity in Software Engineering: A Practitioners Systematic Literature Review. In *2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)* (pp. 145–154).

Fritz, T., & Müller, S. C. (2016, March). Leveraging Biometric Data to Boost Software Developer Productivity. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Vol. 5, pp. 66–77).

Glaser, B. G. (1978). *Theoretical Sensitivity*. CA, USA: Sociology Press.

Glaser, B. G. (1992). *Emergence vs Forcing - Basics of Grounded Theory Analysis*. CA, USA: Sociology Press.

Glaser, B. G., & Strauss, A. L. (1967). *The Discovery of Grounded Theory*. New Jersey, USA: Aldine-Transaction.

Gonçales, L. J., Farias, K., & da Silva, B. C. (2021, August). Measuring the cognitive load of software developers: An extended Systematic Mapping Study. *Information and Software Technology*, *136*, 106563. doi: 10.1016/j.infsof.2021.106563

Gonçales, L. J., Farias, K., da Silva, B. C., & Fessler, J. (2019, May). Measuring the Cognitive Load of Software Developers: A Systematic Mapping Study. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)* (pp. 42–52). (ISSN: 2643-7171)

Gorra, A. (2019). Keep your Data Moving: Operationalization of Abduction with Technology. In *The SAGE Handbook of Current Developments in Grounded Theory.* London, UK: SAGE Publications.

Gulliksen, J., Lantz, A., Walldius, Å., Sandblad, B., & Åborg, C. (2015). *Digital arbetsmiljö, en kartläggning (RAP 2015:17)* (Tech. Rep.). Retrieved from `https://www.av.se/arbetsmiljoarbete-och-inspektioner/kunskapssammanstallningar/digital-arbetsmiljo-kunskapssammanstallning/`

Helgesson, D., Appelquist, D., & Runeson, P. (2021). A grounded theory of cognitive load drivers in novice agile software development teams. In *Unpublished manuscript.* Retrieved from `http://arxiv.org/abs/2107.04254`

Helgesson, D., Engström, E., Runeson, P., & Bjarnason, E. (2019). Cognitive Load Drivers in Large Scale Software Development. In *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering* (pp. 91–94). Piscataway, NJ, USA: IEEE Press. doi: 10.1109/CHASE.2019.00030

Henley, A. Z., & Fleming, S. D. (2016, September). Yestercode: Improving code-change support in visual dataflow programming environments. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 106–114). (ISSN: 1943-6106)

Hollan, J., Hutchins, E., & Kirsh, D. (2000, June). Distributed Cognition: Toward a New Foundation for Human-computer Interaction Research. *ACM Trans. Comput.-Hum. Interact.*, *7*(2), 174–196.

Hollender, N., Hofmann, C., Deneke, M., & Schmitz, B. (2010, November). Integrating cognitive load theory and concepts of human–computer interaction. *Computers in Human Behavior*, *26*(6), 1278–1288.

Hutchins, E. (1995). *Cognition in the Wild.* MIT Press.

Karras, O., Risch, A., & Schneider, K. (2018). Interrelating Use Cases and Associated Requirements by Links: An Eye Tracking Study on the Impact of Different Linking Variants on the Reading Behavior. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018 - EASE'18* (pp. 2–12). Christchurch, New Zealand: ACM Press.

Kelleher, C., & Hnin, W. (2019). Predicting Cognitive Load in Future Code Puzzles. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19* (pp. 1–12). Glasgow, Scotland Uk: ACM Press.

Kelleher, C., & Ichinco, M. (2019, October). Towards a Model of API Learning. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 163–168). (ISSN: 1943-6106)

Kirsh, D. (2000). A Few Thoughts on Cognitive Overload. *Intellectia*(30), 19–51.

Krancher, O., & Dibbern, J. (2015, January). Knowledge in Software-Maintenance Outsourcing Projects: Beyond Integration of Business and Technical Knowledge. In *2015 48th Hawaii International Conference on System Sciences* (pp. 4406–4415). (ISSN: 1530-1605)

Lenberg, P., Feldt, R., & Wallgren, L. G. (2015, September). Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, *107*, 15–37.

Martin, V. (2019). Using Popular and Academic Literature as Data for Formal Grounded Theory. In *The SAGE Handbook of Current Developments in Grounded Theory.* London, UK: SAGE Publications.

Méndez Fernández, D., & Passoth, J.-H. (2019, February). Empirical software engineering: From discipline to interdiscipline. *Journal of Systems and Software*, *148*, 170–179.

Miller, G. A. (1956). The magical number seven plus or minus two: some limits on our capacity for processing information. *Psychological review*, *63*(2), 81–97.

Moody, D. (2009, November). The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, *35*(6), 756–779. doi: 10.1109/TSE.2009.67

Moreno, R. (2010, March). Cognitive load theory: more food for thought. *Instructional Science*, *38*(2), 135–141.

Moseler, O., Wolz, M., & Diehl, S. (2020, September). Visual Breakpoint Debugging for Sum and Product Formulae. In *2020 Working Conference on Software Visualization (VISSOFT)* (pp. 133–137).

Müller, S. C., & Fritz, T. (2016, May). Using (Bio)Metrics to Predict Code Quality Online. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (pp. 452–463). (ISSN: 1558-1225) doi: 10.1145/2884781.2884803

Naur, P., & Randell, B. (1969, January). *Software engineering: Report on a conference sponsored by the nato science committee* (Tech. Rep.). Scientific Affairs Division, NATO.

Nonaka, I., Toyama, R., & Konno, N. (2000, February). SECI, Ba and Leadership: a Unified Model of Dynamic Knowledge Creation. *Long Range Planning*, *33*(1), 5–34. doi: 10.1016/S0024-6301(99)00115-6

Paas, F., Renkl, A., & Sweller, J. (2004, January). Cognitive Load Theory: Instructional Implications of the Interaction between Information Structures and Cognitive Architecture. *Instructional Science*, *32*(1-2), 1–8.

Robson, C. (2002). *Real World Research* (2nd ed.). Malden: Blackwell.

Runeson, P., Höst, M., Rainer, A., & Regnell, B. (2012). *Case Study Research in Software Engineering: Guidelines and Examples.* John Wiley & Sons.

Sadowski, C., & Zimmermann, T. (Eds.). (2019). *Rethinking Productivity in Software Engineering*. Berkeley, CA: Apress.

Sedano, T., Ralph, P., & Péraire, C. (2017, May). Software Development Waste. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (pp. 130–140). doi: 10.1109/ICSE.2017.20

Sedano, T., Ralph, P., & Péraire, C. (2019). Removing software development waste to improve productivity. In C. Sadowski & T. Zimmermann (Eds.), *Rethinking productivity in software engineering* (pp. 221–240). Berkeley, CA: Apress. doi: 10.1007/978-1-4842-4221-6_19

Siegmund, J. (2016, March). Program Comprehension: Past, Present, and Future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Vol. 5, pp. 13–20). doi: 10.1109/SANER.2016.35

Snyder, H. (2019, November). Literature review as a research methodology: An overview and guidelines. *Journal of Business Research*, *104*, 333–339.

Stol, K.-J., Ralph, P., & Fitzgerald, B. (2016, May). Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (pp. 120–131). doi: 10.1145/2884781.2884833

Storey, M.-A., Ernst, N. A., Williams, C., & Kalliamvakou, E. (2020, September). The who, what, how of software engineering research: a socio-technical framework. *Empirical Software Engineering*, *25*(5), 4097–4129.

Sweller, J., & Chandler, P. (1994, September). Why Some Material Is Difficult to Learn. *Cognition and Instruction*, *12*(3), 185–233.

Sweller, J., van Merrienboer, J. J. G., & Paas, F. G. W. C. (1998, September). Cognitive Architecture and Instructional Design. *Educational Psychology Review*, *10*(3), 251–296.

Sykes, E. R. (2011). Interruptions in the workplace: A case study to reduce their effects. *International Journal of Information Management*, 10.

Tavory, I., & Timmermans, S. (2019). Abductive Analysis and Grounded Theory. In *The SAGE Handbook of Current Developments in Grounded Theory.* London, UK: SAGE Publications.

Thornberg, R., & Dunne, C. (2019). Literature Review in Grounded Theory. In *The SAGE Handbook of Current Developments in Grounded Theory.* London, UK: SAGE Publications.

Vessey, I. (1991). Cognitive Fit: A Theory-Based Analysis of the Graphs Versus Tables Literature*. *Decision Sciences*, *22*(2), 219–240. doi: 10.1111/j.1540-5915.1991.tb00344.x

Vessey, I., & Galletta, D. (1991, March). Cognitive Fit: An Empirical Study of Information Acquisition. *Information Systems Research*, *2*(1), 63–84. doi: 10.1287/isre.2.1.63