Exploring Grounded Theory Perspectives of Cognitive Load in Software Engineering

Daniel Helgesson

Licentiate Thesis, 2021
Department of Computer Science
Lund University

ISBN 978-91-7895-817-7 (printed version)
ISBN 978-91-7895-818-4 (electronic version)
Licentiate Thesis 2, 2021
ISSN: **1652-4691**

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: daniel.helgesson@cs.lth.se
WWW: `http://cs.lth.se/`

Printed in Sweden by Tryckeriet i E-huset, Lund, 2021

Abstract

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

Aim: The aim of these studies is to increase the understanding of the impact of cognitive load in software engineering. Our ultimate goal is to thereby to reveal opportunities to make software engineering tools more efficient for companies and compelling to developers.

Method: We construct and synthesise knowledge, using grounded theory ethnography and abduction, from our empirical observations and literature on cognitive load in software engineering, with *cognitive load theory* and *distributed cognition* as stepping stones.

Result: We present models 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. In addition, the second model provides an explanation on cognitive load associated to merge operations and version control in *agile software development*.

Conclusion: We intend to use this conceptual model as a starting point for the design of software engineering tools, methods and organisational structures to improve efficiency and developer satisfaction by reducing the cognitive load.

SPIRITUAL ROLE MODELS & PHILOSOPHICAL INSPIRATION

Two historical figures play an important role in the research reported on in this thesis, Martin Heidegger and Robert Oppenheimer. Oppenheimer was my role model as pre-teen, and I designed my first nuclear reactor at age fourteen. While unrelated, during the Nietzsche-fueled *Sturm & Drang*-period of my life (age seventeen) I contemplated a vulgarised Heidegger maxim in along the lines of *confrontation with death is essential for an essential life* while in parallel pondering over reality as a four dimensional time/space nexus. When I stumbled on concept of *Ba* early on in this research project its' association to Heidegger prompted me to reacquaint myself with him, and my yoga practices brought me back to Oppenheimer by way of Bahavad Gita.

– 'Time is not a thing, thus nothing which is, and yet it remains constant in its passing away without being something temporal like the beings in time.'

– 'True time is four-dimensional.'

– 'If I take death into my life, acknowledge it, and face it squarely, I will free myself from the anxiety of death and the pettiness of life – and only then will I be free to become myself.'

M. Heidegger

– 'Both the man of science and the man of action live always at the edge of mystery, surrounded by it.'

– 'It worked!'

– 'Now I am become Death, the destroyer of worlds.'

R. Oppenheimer

It made all the difference.
Namaste.
Lund March 2021

ACKNOWLEDGEMENTS

The guardian angels and muses, without whom I would probably not be alive – Aske, C.J., Y.S., A.H., F.A. (I owe you guys forever).

Fellow researcher – D.A. (it was truly a blast).

The supervisors, without whom this work would not have been carried out – P.R., E.E., E.B. (my guess is, that it was, on occasion, somewhat of a challenge).

Librarians – A.K. & O.H. (while I have acquired some basic notion on how to execute research, those pesky databases remain tricky...).

Fellow ph.d.-students – H.M., S.R., Q.S. (for numerous interesting discussions and oft needed LaTeX-help).

The SERG group, and the department of Computer Science.

The work in this thesis is dedicated to bullied children, and children battling chronic disease world wide. As the odds always will be stacked against you, the only way forward is to walk tall, kick ass, resist and bite – hard. Besides, nobody likes a quitter...

Résiste et Mords![1]

Daniel Helgesson, Lund, March 2021

[1]In May 1940 as the Belgian border was breeched, communications broke down. A rifle company (roughly 40 rifles strong) failed to capture the message of general retreat and as a consequence adhered to the previously received general order (i.e. to hold the border at any cost). The company held the line for 18 days against the *Ghost division* under the command of Erwin Rommel, until they eventually ran out of ammunition. Rommel later lamented that: '...these were not men, they were green wolves...' (or possibly, 'wolves in green clothes' – sources vary in translation). Said company belonged to the 2ème Regiment de Chasseurs Ardennais, whose motto remain *Resist and Bite!* to this day. While their insignia is a wild boar, they are commonly referred to as *The Green Wolves*. Of Wallon lineage (and equipped with a corresponding complexion that cost me dearly during my school years) and having battled chronic autoinflammatory issues since age 18, I hold the resilience displayed by these riflemen, and their motto, not so much a suggestion as an imperative. Gloria fortis miles.

LIST OF PUBLICATIONS

This thesis consists of an introduction and a compilation of three papers. The introduction gives an overview of the research topic, and it describes the papers and contributions briefly. Paper I has been formatted in this thesis template without additional changes from the original publications.

Publications included in the thesis

I **Cognitive Load Drivers in Large Scale Software Development**
Daniel Helgesson, Emelie Engström, Per Runeson, Elizabeth Bjarnason
CHASE, 2019.
DOI: 10.1109/CHASE.2019.00030

II **A Grounded Theory of Cognitive Load Drivers in Agile Software Development**
Daniel Helgesson, Daniel Appelquist, Per Runeson
DOI: Working manuscript

III **Grounded Theory Perspectives of Cognitive Load in Software Engineering**
Daniel Helgesson, Per Runeson
DOI: Working manuscript

Contribution statement

All papers included in this thesis have been co-authored with other researchers. The authors' individual contributions to Papers I-III are as follows:

Paper I
First author, Daniel Helgesson (DH), and fourth author, Dr. Elizabeth Bjarnason (EB), designed the the initial interview guides and conducted the first three interviews. The first round of analysis was conducted by DH, Dr. Emilie Engström (EE) and Prof. Per Runeson (PR). Following two additional interviews by DH, a second round of analysis was conducted by DH and EE. The resulting taxonomy was developed by DH, EE and PR. All interviews were transcribed by DH. The major part of the paper was written by DH, section *Method* was written by EE, section *Introduction* was written by PR. All authors contributed to reviewing and editing the final submission of the paper. The published version, included in this thesis, was cut from 8 to 4 pages by DH.

Paper II
First author, Daniel Helgesson (DH), designed the study. DH and second author Daniel Appelquist (DA) created the data set from observations and other sources. Interviews were conducted by DH. Focus group sessions were conducted by DH, DA and third author Prof. Per Runeson (PR). Open coding was executed by DH and DA. In depth analysis, consisting of focused coding and theoretical coding, was executed by DH, using memos. Literature review was conducted by DH. The main part of the paper was written by DH (based on memos), with the exception of section *Introduction*, written by PR. Illustrations were made by DH in ink on paper and later transferred to electronic form by PR. DH and PR collectively reviewed and edited the version of the manuscript presented in this thesis.

Paper III
First author Daniel Helgesson conducted the abductive analysis and theoretical coding, largely using memos. Literature review on Cognitive Load Theory was conducted by DH with the help from librarian Andreas Karman in formulating queries and trouble shooting results. The main part of the paper was written by DH (based on memos), with the exception of section *Introduction*, written by PR. Illustrations were a joint effort between DH and PR. DH and PR collectively reviewed and edited the version of the manuscript presented in this thesis.

CONTENTS

INTRODUCTION

1 Software Engineering and software engineering

The meaning of Software Engineering is dual. One meaning is literal, *software engineering* is the activity of engineering software – while the other, *Software Engineering* (SE) is the scientific study of software engineering activities, methodologies, development tools etc. The term *Sofware Engineering* was coined during the first NATO conference on the matter [43] in 1968. The main raison d'etre and mission of the *Software Engineering* research community is to provide practitioners with knowledge regarding software engineering methodologies, tools and other relevant dimensions. But despite its name, SE is different in regards to most other disciplines [6] in so motto that it is largely unbound by physical laws and exists in an artificial environment.

The human centric aspects of SE was acknolwedged at an early stage – *personel factors* was one dimension discussed at the initial NATO conference [43]. Further, Bertelsen [6] described one major difference between software engineering and traditional engineering activities as 'the sociocultural constitution of software', another the (extremely) rapid progress in technology[2]. This reasoning was a few years later further extended by Shaw [62]. Yet, despite a scientific understanding that software engineering is a people centric, sociocultural and sociotechnical phenomenon technical aspects of software engineering are considerably more extensively studied, and published, than *softer* [38] [67], more human centric, dimensions of software engineering. Further, modern software development is not only people centric, it is also high paced, iterative, information dense and dependent on a large number of integrated and stand alone software tools and information systems (e.g. IDE, version control, defect. handling, email etc).

[2]https://en.wikipedia.org/wiki/Moore%27s_law

1.1 Cognitive dimensions of software engineering

Software engineering, as a phenomenon, is cognitively instensive [53], and the cognitive capacity of humans has since the 1950s generally been accepted as finite (and quite limited) [41] . Further, all human task resolution, problem solving and information processing inherently induce *cognitive load*, mental effort, on the human mind. So, in essence, software engineering is a people centric sociotechnical endeavour of humans operating tools iteratively under information dense conditions and commonly under sharp time constraints, where it is not farfetched to view the human mind as a cognitive bottleneck.

1.2 Cognitive load, cognitive load theory and distributed cognition

Cognitive load can be loosely translated into *mental effort* or *mental strain*. It has since Millers seminal paper [41] been generally accepted that the human working memory (or bandwidth for information processing) is finite and rather limited. This is expanded on in *cognitive load theory* [68] which states that *extraneous* (unnecessary) cognitive load should be removed from instruction materials in order to free up more cognitive resources for the actual task at hand. In the first study presented in this thesis we coined the term *cognitive load driver* to describe the root causes of unnecessary cognitive load in a software development context. The consequences of cognitive overload is well known (it is actually the reason human factors engineering was created - to figure out why perfectly functioning pilots insisted on crashing perfectly functioning airplanes, the US Air Force called in experimental psychologists [71]).

Many software engineering task are inherently cognitively loaded (i.e. with a high *intrinsic cognitive task load*) so freeing up, or minimising, *extraneous cognitive load*, unrelated to the actual task, will ultimately allow to direct more effort on the actual task at hand, rather than waste it on matters unrelated to the task at hand. In all likelihood it will also make developers less stressed and error prone.

It is also easily observable that software engineering is a distributed, or collective, endeavour where several people work together solving cognitively loaded tasks, essentially a form of distributed cognition. *distributed cognition* is a subdiscipline of studies of cognition that rejects one of the traditional cornerstones of cognition, 'that cognitive processes such as memory, decision making and reasoning, are limited to the internal mental states of an individual' [28]. Instead it argues that the social context of individuals and artefacts form a cognitive system transcending the cognition of each individual involved [21], i.e. 'a cognitive system [that] extends beyond an individual's mind' [39].

1.3 Content

This thesis explores cognitive load in a software engineering context, using *cognitive load theory* and *distributed cognition* as scientific lenses. It includes three papers and is organised as follows:

2. Overall research goals – this section outlines the overall research goals and states the research tasks presented in this thesis.
3. Contribution – this section describes the scientific contribution of this thesis.
4. Concepts – this section describes, and clarifies, the concepts and constructs used and explored in this thesis.
5. Related work – this section gives a brief overview of related work on cognitive load in a software engineering context.
6. Epistemological stance – this section provide an epistemological discussion of the contents of the thesis.
7. Methodology – this section describes the research methods used in this thesis.
8. Findings – this section provides an overview of the findings from the papers included in this thesis.
9. Conclusions – this section contains a conclusion of the thesis.
10. Future research – this section outlines an overview of future research directions.
11. Papers – the chapters *Paper I*, *Paper II* and *Paper III* contain the papers included in this thesis.

2 Overall research goals

The research described in this thesis has been explorative with an over-arching research goal of *exploring cognitive load* in a software engineering context and *exploring methodology and scientific lenses suitable for investigating* said phenomena in said context. As we noted early on that research efforts on 'softer' issues in the SE community were marginal compared to more technical aspects, we decided to pursue an open ended explorative research goal, rather than a preformulated research goal.

The explorative research goal has since been broken down in different research tasks):

1. To explore cognitive load in a software engineering context.

2. To empirically describe cognitive load in a software engineering context.
3. To explore methodologies suitable for investigating cognitive load in a software engineering context.
4. To explore the use of distributed cognition in SE.
5. To explore the use of cognitive load theory in SE.
6. To propose models for reasoning on cognitive load in a software engineering context.

3 Contribution

The scientific contributions of this thesis consist of:

1. An initial exploration and taxonomy of cognitive load drivers in SE, dervied from an exploratory case study (Paper I).
2. An intial literature survey of cognition in software engineering (Paper I).
3. A model[3] describing cognitive load drivers in agile software development derived from field observations (paper II).
4. A literature survey of the use of distributed cognition in SE (Paper II).
5. A model for reasoning on how different forms, or *perspectives*, of cognitive load affects productivity and cognitive sustainability in software engineering, derived from from the aggregated result of Papers I & II and the results reported by Sedano et al. [53](Paper III).
6. A literature survey of the use of cognitive load and cognitive load theory in SE (paper III).

4 Concepts

This section describes, and clarifies, the concepts, constructs and terminology used in this thesis. It is divided into two subsections – *Research concepts* describes concepts and constructs in the research area described in this thesis, while *Empistemological and methodological concepts* describe overall epistemological and methodological concepts used in this thesis.

[3]In order to avoid epistemological confusion we use the construct *model* – the corresponding grounded theory construct is *substansive theory* [11]

4.1 Research concepts

Cognition

Oxford English disctionary defines *cognition* as: 'The mental action or process of acquiring knowledge and understanding through thought, experience, and the senses.'[4]

In addition *cognitive science* (commonly referred to as *cognition*) is an intradisciplinary scientific field where the ambition is to explore, describe and explain the function of the human mind.

Cognitive load

Cognitive load as a phenomenon, can be loosely translated into *mental effort*. It is inherent in all forms av cognitive work.

Cognitive work

The meaning of *cognitive work* is dual. The first meaning is literal (i.e. doing work that is cognitively loaded such as software engineering tasks), while the other is more philosophical and/or physical. Just as *load/power* in the physical sense is the time derivative of *work/energy* (or vice cersa *work* is the time integral of *load*). While this reasoning is an analogy, it none the less provides a credible explanation of why prolonged exposure to cognitive overload is unhealthy and as a consquence should be avoided. Sedano et al. [53] highlight that most tasks in software development are *cognitively intense*.

Cognitive bandwidth

As mentioned previously it has since the mid fifties, in the wake of Millers seminal paper *The magical number seven, plus or minus two: some limits on our capacity for processing information* [41], been generally accepted that the human working memory and bandwidth for information processing, or *cognitive bandwidth*, is finite and distinctly limited.

Cognitive load drivers

In Paper I we coinded the term *cognitive load driver* to describe causes, or sources, of unnecessary cognitive load. This largely corresponds to *extraneous cognitive load* in cognitive load theory, but is used to identify the actual cause.

[4]https://www.lexico.com/definition/cognition

Cognitive productivity & waste

In cognitive load theory *extraneous* (i.e. *irrelevant*) can be considered *waste*. It is cognitive resources wasted outside of the cognitive task at hand. Reducing cognitive waste will, in all likelihood, yield a decrease in *cognive load* and a corresponding increase in terms of *cognitive productivity*.

Sedano et al. discuss extraneous cognitive load in relation to productivity (or actually, *waste*) in software development [53]. Increase in *cognitive productivity* would be one of the obvious consequenses of reducing *extraneous cognitive load* (or, *cognitive waste*) in software engineering.

Cognitive overload

The concept of *cognitive overload* is the cognitive equivalent of *electrical overload*, i.e. a higher cognitive load than what is sustainable. If we see *cognitive overload* as momentary the longitudinal consequence would be *cognitive drain*. The phenomenon itself, and its consequences in a workplace setting are throughly described by Kirsh [35].

Cognitive sustainability

When exploring cognitive load and thoughts on cognitive productivity, the term *cognitive sustainability* was constructed. We know that longitudinal exposure to cognitive overload not only affects productivity and learning abilities, but also results in psychological and physical health issues[5]. As a consequence the concept of *cognitive sustainability* could be used when attempting to formulate guidelines for a healthier software engineering environment from an ergonomic as well as from a cognitive ergonomic perspective.

Cognitive amplification

In the initial phases of the research described in this thesis the concept of *cognitive amplification* was explored, when trying to describe the positive function of software engineering tools. It was used to allow for a reasoning on the ratio (i.e. actual *gain*) between positive aspects (i.e. cognitive amplification) and negative aspects (i.e. cognitive load) of a software tool. The earliest mentioning of the concept encountered in this research is provided by Card et al. [10], and is exemplified by the use of pen and paper when adding or subtracting large numbers. Kirsch describes the same phenomenon when using gridded paper when adding/subtracting large numbers [35].

[5]I hold the concept of *burnout* as self evident, and will not back it with a reference.

Perspectives

When synthesising the results from Papers I & II with the findings presented by Sedano et al. [53] we coined the term *perspectives* (of/on cognitive load). The concept was the result of abductive reasoning on cognitive load theory and critique on cognitive load theory. We realised that cognitive load, despite the actual cognitive load driver, can be described in several different ways and that it is hard to define in an orthogonal fashion. As a consequence we used the construct *Perspectives* to signify that there is often an overlap between the different aspects of cognitive load in a software engineering context.

The temporal domain

The temporal domain is a construct that we have used in an attempt to describe that there is a temporal dimension of cognitive load in software engineering. The temporal domain was encountered early on in the research process. We initially noted temporal dimensions of cognitive load drivers in Paper I (i.e. absence of version control/version history and strategies for finding information about events that took place in the past. We encountered reasoning on the temporal nature of cognition in the Heideggerian time/space nexus concept of *Ba* provided by Nonaka et al. [44] and further in distributed cognition [32].

In Paper II we noted another temporal dimension of cognitive load, *temporal synthesis*, exemplified by merge operations that can be described just as a cognitive fusion or *synthesis* of two (commonly) chunks of code, commonly written by different persons at different *times*. We described this as a *sensitizing concept* in Paper III.

Information & instructions

In parallel with the temporal dimensions of cognitive load in software engineering we also noted that the nature of information in software is dual. It can be viewed as *information* or *instructions*. Regardless if we choose to view it as information or instructions, it can be divided into two categories, namely *essential information/instructions* (i.e. source code or executable code) and a *meta*-perspective of *essential information/instructions* – i.e. information about source code (e.g. comments, commit messages, design documentation etc.) or information about executable code (e.g. issues and error reports).

4.2 Epistemological and methodological concepts

Theory & model

In this thesis I rely on grounded theory as a methodogical approach. In Paper II a *substansive theory* [11] was generated, while in Paper III a set of constructs were generated. As the construct *theory* conveys different meanings in different scientific disciplines and epistemological traditions we have tried to systematically use the construct *model* [65] in order to avoid epistemological confusion. In the context of this thesis *model* should be interpreted as a set of constructs and relations that can be used to analyse the present and/or to predict future outcomes.

Postpositivism

In the research described in this thesis we state our epistemological position as *pragmatic postpositivism*, i.e. we assume that there is a reality that can be objectively (albeit imperfectly) described using qualitative *OR* quantitative research instruments and methodologies. As with *theory* the concept of *postpositivism* seems to convey different meanings in different scientific traditions – a reflective paper on the study described in Paper II was submitted in a course on qualitative methodology and was ultimately failed on epistemological grounds (i.e. it was not considered postpositivist). In this thesis I draw on the definition provided by Robson [48], and note that there are phenomena in sociotechincal contexts that are hard to investigate and describe in a quantitative positivistic manner. None the less, I hold it as self evident that the phenomena do actually exist and their existance have consequences.

In order to avoid further epistemological confusion it should also be noted that I use the word *pragmatic* in the literal sense, not the epistemological [3] or philosopical sense (albeit they are all quite similar in this context). Our research interest lies in producing relevant knowledge for the Software Engineering research community and for software engineering practitioners, not epistemological bloodsports. I think that the nature of the phenomena/-on under study should determine the epistemological and methodological approach to inquiry, not the other way around.

Distributed cognition

As described earlier, *distributed cognition* is a sub-discipline of studies of cognition in which one of the traditional cornerstones of cognition, 'that cognitive processes such as memory, decision making and reasoning, are limited to the internal mental states of an individual' [28] is questioned and rejected. Instead it argues that the social context of individuals and artefacts forms a cognitive system transcending the cognition of each individual involved [21] – that is: 'a cognitive system [that] extends beyond an individual's mind' [39]. The concept was pioneered by Hutchins who studied the cognitive activities on the navigation bridge of US naval vessels [34].

Hollan, Hutchins and Kirsh extended distributed cognition into the realm of human computer interaction as well as to some extent into Software Engineering, stating that a distributed cognitive process (or system) is 'delimited by the functional relations among the elements that are part of it, rather than by the spatial colocation of the elements', and that as a consequence several interesting aspects can be observed, namely that: '[a] cognitive processes may be distributed across members of a social group[;] [b] cognitive processes may involve coordination between internal and external (material or environmental) structure [and] [c:] processes may be distributed through time in such a way that the products of earlier events can transform the nature of later events.' (reformatted but verbatim). [32]

Cognitive load theory

As previously stated, in *cognitive load theory* Sweller (and others) states that the human working memory is finite and limited, and that exposure to cognitive overload will impair or inhibit learning as well as problem solving [68]. Stemming from educational research cognitive load theory divides cognitive load into three principal components, namely *intrinsic*, *extraneous* and *germane* cognitive load. *Intrinsic* cognitive load refers to the inherent cognitive load of processing/solving the task at hand and *extraneous* cognitive load to cognitive load induced on the mind by the environment or by how the information, or task, is presented. *Germane* cognitive load refers to the mental effort used when forming, constructing and automating problem solving schemas and processing of information. cognitive load theory states that reduction of the *extraneous* (or *irrelevant*) cognitive load will reduce the burden of the working memory and therefore is associated with more efficient learning and problem solving.

Grounded theory

Grounded theory [11] (GT) is an inductive approach to generating descriptive and predictive theory from (mainly) qualitative data, that was originally described by Glaser & Strauss in *The Discovery of Grounded Theory* [25]. With grounded theory, Glaser & Strauss provided a powerful methodological framework for inductive reasoning based on qualitative data that opposed to the quantitative and hypotetico-deductive methodologies and/or paradigms that at the time (mid sixties) were dominant in social sciences. According to Charmaz, Glaser and Strauss 'aimed to move qualitative inquiry beyond descriptive studies into the realm of explanatory theoretical frameworks, thereby providing abstract, conceptual understandings of the studied phenomena'[[11].

While there are multiple versions, or currents, of grounded theory existing in parallel, there is a set of core principles that unify them. Charmaz & Mitchell [12] provide the following set of core principles (or *features*) for all variants of grounded theory:

1. Simultaneous data-collection and analysis.
2. Pursuit of emergent themes through early data analysis.
3. Discovery of basic social processes within the data.

4. Inductive construction of abstract categories that explain and synthesize these processes.

5. Integration of categories into a theoretical framework that specifies causes, conditions and consequences of the process(es).

One important methodological information processing tool in the grounded theory framework is *memoing* - the researcher uses *memoing* to write *memos* describing and processing information and concepts central to the research. The writing style is *fast*, *free* and *loose* in the spirit of the purpose, i.e. to process observations and information – not to produce paper grade prose.

Ethnography

Just as with grounded theory, ethnography stems from social sciences. Meaning *recording the life of a particular group* [55] it is rooted in ethnology and anthropology, with modern ethnography stemming from Malinowski's research of indigenous populations during WW I. The central tenet of modern ethnography is to *describe another culture[or group] from a member's point of view* [55].

Sharp et al. [55] highlight two fundamental perspectives that make ethnography differ from other means of qualitative inquiry used in Software Engineering, the *empathic perspective* i.e. the researcher gaining insight from the group being studied *as seen through the eyes of those studied* forcing the researcher to *bracket* his or her presuppositions, and further the *analytical focus/stance* allowing the researcher not only to capture *what is done* but also, more fundamentally, *why* it is done. In addition two other prominent features of ethnography is described, *the ordinary detail of life as it happens* and *thick descriptions*.

5 Related work

5.1 General cognition in SE

As mentioned earlier the first report on *Software Engineering* mentioned *personel factors* as one relevant aspect of SE [43]. Since then various dimensions of cognition related to sofware engineering have been explored.

Siegmund [64] provide an overview of past, present and future aspects of *program comprehension* efforts. Blackwell et al. [7] provide an historical overview of *50 years of psychology in Programming*. Relevant examples of early approaches to describe cognition in software engineering include Lagherty & Lagherty [37] and Koubeck et al. [36].

An updated mapping study on *measurements on cognitive load in software engineering* is provided by Gonçales et al. [26]. One example of a study focused on cognitive measurements is provided by Fritz & Müller [22].

Version control from a user perspective has been studied by Church et al. [14], de Rosso & Jackson [18] and Perez & de Rosso et al. [45].

5.2 Grounded theory in SE

Stol & Fitzgerald [65] provide a discussion on the need of theory generation in SE, in general, as well as suggestions for a way forward.

Specific guidelines regarding to grounded theory in SE and a critical review of the state of GT research in SE is provided by Stol et al. [66], while Hoda et al. [29] provide additional guidelines on GT in SE.

Adolph et al. [1] and Sedano et al. [52] provide relevant reflections on the experience of GT research in SE context.

Further, I will let work by Adolph et al. [2], Baltes & Diel [5], Sedano et al. [53] [51] [54] and Hoda et al. [30] [31] serve as notable examples of GT research in SE.

5.3 Ethnography in SE

Guidelines on conducting ethnographic research in SE context is provided by Sharp et al. [55] and by Robinson et al. [47].

An updated example of ethnography in SE is provided by Zaina et al., who used ethnography with distributed cognition as theoretical underpinning when investigating *UX* (user experience) information and artefacts in agile software development teams [73].

Seminal examples of ethnograpical inquiries include Sharp & Robinson [57] [**?**] and Sharp et al. [60].

5.4 Cognitive ergonomics

In Paper I we compared our findings against a taxonomy on general cognitive work environment issues provided by Gulliksen et al. [27]. In addition Sykes [69] provides an inquiry on the cause and effect of interruptions in a software engineering context. Kirsh [35] provides a reasoning on the cause and effect of cognitive overload in a general workplace setting. In addition, several chapters in 'Rethinking Productivity in Software Engineering' [50] border on cognitive ergonomics.

5.5 Distributed Cognition in SE

In Paper II we used distributed cognition as a scientific lens for our observations, and further charted the impact and legacy of distributed cognition in SE. Despite the fact that the theory of distributed cognition[6] was suggested as a fruit-

[6]The use of *distributed cognition* in lower case is intentional – the meaning should be interpreted as cognitive processes that are distributed.

ful approach for investigating and explicating phenomena related to software engineering several decades ago [32] (Flor and Hutchins specifically studied pair-programming [21] from a distributed cognition perspective as early as 1991), few examples of actual software engineering studies using distributed cognition as a scientific lens exist.

In 2014 Mangalaraj et al. [39] highlighted Sharp and Robinson [58], Hansen and Lyytinen [28] and Ramasubbu et al. [46] as *the few notable exceptions* of extant software engineering research utilising Distributed Cognition. To this list we would like to add Walenstein [70], a recent study by Buchan et al [9] and other works by Sharp et al. [57] [60] [61] [56] [59].

5.6 Cognitive load in SE

In Paper III we synthesised our findings from papers I & II, with the findings of Sedano et al. [53], and further used cognitive load theory as a base for our synthesis. We also charted the use of *cognitive load* in extant SE literature. While not specifically using cognitive load theory as scientific lens, Sedano et al. [53] used the construct of *extraneous cognitive load* in a grounded theory study classifying software development waste. Measuring cognitive load is quite popular, e.g. Fritz & Müller [22], and a recent systematic mapping study specifically on measuring the cognitive load of software developers is presented by Gonçales et al. [26].

6 Epistemological stance

At the time of writing it seems prudent to highlight that all knowledge produced qualitative research is inherently constructed[7], and that I take a pragmatic postpositivist [48] stance in terms of epistemology.

The aim of the research reported on in this thesis is to provide knowledge around phenomena that do occur (i.e. exist) in the software engineering realm. Despite the knowledge being constructed, the phenomena do, objectively, exist, albeit in an artificially created environment. My ambition is to investigate these phenomena and their consequences as neutrally and objectively as possible, in order to firstly provide means of understanding the cause and effect of these phenomena and secondly to provide advice on how to lessen the impact of these phenomena.

My take is that the epistemological position of a study should reflect the nature of the phenomenon/-a under study. When describing *(*cognitive load drivers), their *impact* and their *consequences* an objectivist/postpositivist stance appears suitable (to me). When descibing how people *experience* phenomenon/-a a constructivist/-postpositivist stance is in all likelihood more suitable.

In regard to the generalisability of qualitative knowledge and single case studies, I humbly point to Anzai & Simon [4] – '*It may be objected that a general*

[7]I hold this as so self evident that I do not see the need for a reference backing this statement

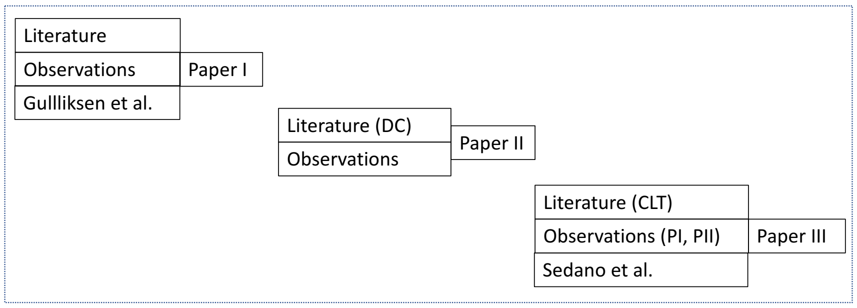

Figure 1: The relation between the different studies used, and papers included, in this thesis.

psychological theory cannot be supported by a single case. One swallow does not make a summer, but one swallow does prove the existence of swallows. And careful dissection of even one swallow may provide a great deal of reliable information about swallow anatomy.'

7 Methodology

This section describes the research methods used in this thesis. Thus far the research has largely been explorative, qualitative and has relied on case study methodology and grounded theory. The relation between the papers, and the corresponding studies, included in this thesis are shown in Figure 1.

7.1 Paper I

The first study was primarily aimed at establishing the relevance of cognition as a scientific lens in software engineering, and further to provide empirical evidence of cognitive load drivers in SE. As a consequence the study was designed as an explorative case study [49], using a flexible design, including an open ended literature immersion to find, and explore, scientific lenses relevant for sofware engineering from cognition.

The field study consisted of a dataset of semi structured interviews and thematic analysis [8] was used to analyse the data set. The design of this study was not formulated as grounded theory per se, as previously mentioned it was designed as an *explorative case study*. While not positioned as grounded theory it did contain a considerable degree of grounded theory practices: it was *explorative*, it used *initial research goals* rather than preformulated research questions, it featured *it-*

erative data collection and analysis, as well as *open coding* and we returned to the field for additional data after the first round of analysis for *theoretical saturation*.

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* [11]. The literature immersion provided the insight that softer/humane aspects are not a well researched area in software engineering, and suggested distributed cognition and cognitive load theory as scientific lenses and stepping stones for reasoning and analysis of cognitive load. The taxonomy was compared to that presented by Gulliksen et al. [27] for validation purposes.

7.2 Paper II

The second study, a grounded theory ethnographic study was aimed at understanding cognitive load drivers from the novice point of view using Distributed Cognition as a scientific lens. The dataset was crystallised from a multitude of data sources (field notes, focus groups, semi structured interviews, questionnaires as well as written reflections). Further the study also involved three field experiments. After the first round of coding we returned to the field for further interviews in order to achieve theoretical saturation. Flexibility was highlighted in the study design, and is of high importance in the ethnographic approach [55] [20]. Charmaz [11] was used as the main grounded theory research guidelines, while additional works of Glaser [23] [24] and Stol et al. [66] were consulted as complementing perspectives.

In the study analysis was executed in three stages; open initial coding, followed by focused coding and theoretical coding. The analysis relied heavily on memoing [11] [23] [66].

7.3 Paper III

The third study, an abductive synthesis of the first study, the second study and the findings of a third grounded theory case study on productivity waste [53] used cognitive load theory as scientific lens. The findings of the studies were used as 'sensitizing concepts' [11]. This study relied on grounded theory, specifically abductive reasoning as proposed by Martin [40]. In order to chart the use of cognitive load theory in SE it featured a small literature study. In addition to Martin, we also relied on reasoning on qualitative synthesis provided by Cruzes & Dybå [15] [16].

8 Findings

All three papers are aimed at the overall research goal and the first and second research tasks, to *explore* and *empirically describe* cognitive load in a software engineering context. The reserach tasks adressed by the individual papers are listed in section 2.

8.1 Paper I

Paper I investigates how *cognitive load* affect the productivity of software engineers. The interviews allowed us to create an initial taxonomy of *cognitive load drivers*, causes of cognitive load, in software engineering, while at the same time revealing that when a much disliked software system at the case company was replaced, it was on account of license costs – not user dissatisfaction. We further noted in the saturating interviews that a considerable amount of the missing functionality lamented on by users (e.g. version control and search functionality) in the two systems under study was actually supported by the respective platform, but switched of in the proprietary configuration used by the case company.

The *cognitive load drivers* found gravitated around three clusters, namely *tools*, *information* and *work, process & structure*. Ultimately the taxonomy was compared to that of *cognitive work environmental issues* from a more general digital work environment perspective by Gulliksen et al [27]. The clusters are shown in Figure 2.

Two of the interviews suggested that there is a temporal dimension of cognitive load in software engineering. The initial taxonomy created, and the literature findings (e.g. *distributed cognition*, *cognitive load theory* and *expertize* as described by Chi et al. [13], largely correspond to *sensitizing concepts* in grounded theory.

All in all, the findings of the paper suggest that:

1. Cognitive dimensions[8] of software engineering were not thoroughly investigated by the SE research community.

2. Cognitive dimensions of software engineering, and end user satisfatcion of software development tools, were not really considered as important at the case company.

3. Further investigation of cognitive load drivers in a software engineering context would benefit the software engineering community of practitioners as well as the SE research community.

4. The suitability of distributed cognition and cognitive load theory as scientific lenses for further research on cognitive load drivers in a software engineering context.

[8]Since the conjunction of the words *cognitive* and *dimensions* appears to be considered proprietary by some reviewers it seems prudent to highlight that the use in this context is literal.

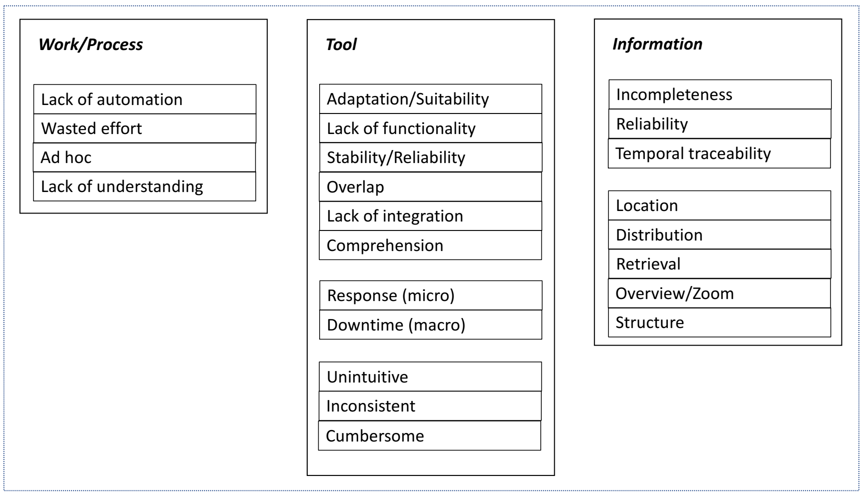

Figure 2: Visualisation of the different clusters of *cognitive load drivers* found in Paper I.

8.2 Paper II

Paper II investigates how cognitive load drivers in general and temporal cognitive load drivers affect the productivity in *agile software development teams* from the *novice point of view*, using grounded theory ethnography, and multimethod data collection, as main methdodology and distributed cognition as a scientific lens. The study design was extremely flexible and open ended as we had no idea of what we would encounter in the field study. We observed four ten person teams consisting of sophomore computer engineering students working as a team one day per week for seven weeks.

Using iterative open, focused and theoretical coding of the dataset we created a (substansive) grounded theory that describes how version control, branching and mere operations make up considerable cognitive load drivers as a consequence of the intrinsics of Agile software development and lacking tool support/tool integration. The constructs and their causal relation are shown in Figure 3.

We further noted that grounded theory and grounded theory ethnography are indeed suitable methodological apporaches for observing and describing human centered phenomena in software engineering, and that distributed cognition serves as a suitable scintific lens for investigating these phenomena, especially in a team (i.e. distributed) context.

Literature reviews revealed that despite its' promises, the impact of distributed

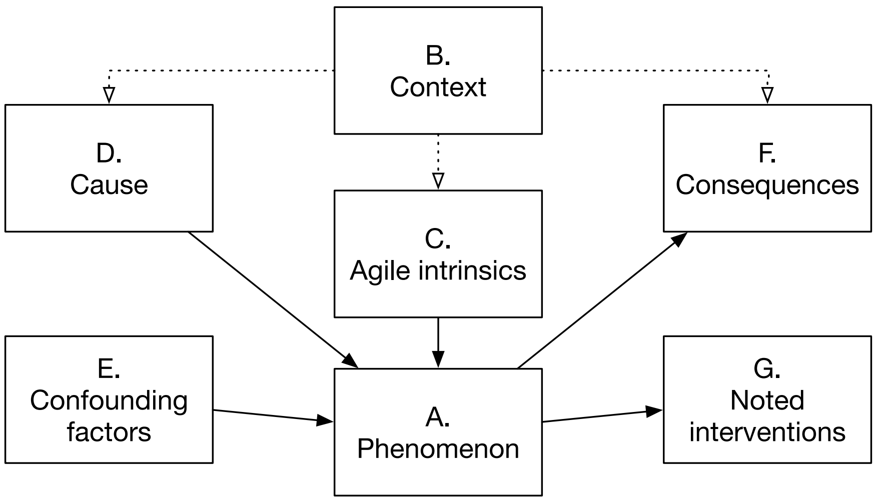

Figure 3: Conceptual model of the causal and consequential dimensions in regards to version control, branching and merge operations encountered in the projects. From Paper II.

cognition in SE research has thus far been marginal. Further, in the limited literature review on version control, branching and merge operations the result was very meagre.

All in all, the findings of the paper suggest that:

1. When not supported properly by tooling and processes, version control and merge operations cause considerable problems in agile software development.

2. There seems to be a considerable research gap in regards to version control and merge operations in extant SE literature.

3. Distributed cognition is indeed a suitable scientific lens for analysing team oriented software development.

4. Grounded theory ethnography is indeed a suitable methodology for investigating some phenomena in a software engineering context.

5. Further investigation of cognitive load drivers in a software engineering context would benefit the software engineering community of practitioners as well as the SE research community.

Table 1: Cognitive load perspectives grouped by cognitive load theory components and association to the data sets. From Paper III.

CLT component	**Perspective**	**PI**	**PII**	**[53]**
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	

8.3 Paper III

Paper III contains a qualitative synthesis of three studies – Paper I (PI), Paper II (PII) and the findings from a GT study on *software development waste* by Sedano et al. [53]. The paper relies on qualitative synthesis as suggested by Cruzes & Dybå [15] [16] using grounded theory abduction [40] as main methodology using cognitive load theory as a scientific lens.

When studying critique on cognitive load theory [33] [17] [42] [19] it became apparent that the different components of cognitive load, as described by cognitive load theory [68], are very hard to describe in a orthogonal taxonomical fashion as they are to some extent overlapping.

Using grounded theory abduction with a qualitative data set consisting of the conjuction of PI, PII and [53] combined with literature we constructed a model for reasoning on cognitive load, *perspectives on cognitive load*, in a software engineering context.

The different *perspectives* consist of *task*, *environmental*, *information*, *tool*, *communication*, *structural*, *interruption* and *temporal*. During the analysis we additionally noted several sensitizing concepts, that will feed into the future research proposed by Paper III and this thesis. The *perspectives*, their relation to the components of cognitive load theory and their relation to the different data sets/papers are shown in Table 1.

In the literature review we noted that while *cognitive load* is a phenomenon that is known in the SE research community, it is not throughly researched. The research that has been conducted is largely quantitative [26] using various means of sensors for data collection and statistical inference as mean of analysis (e.g. Fritz & Müller [22]). We noted that outside of program comprehension, the reasoning on cognitive load is small in the SE research community. We further noted that the impact of cognitive load theory, as such, in SE appears as small.

9 Limitations

This section aims at adressing limitations of the research described in this thesis. As each paper has a section on *Threats to validity* or *Limitations* this section taims to adress the limitations of the general thesis rather than the individual papers. It adresses limitations and validity from a GT perpsective using evaluation criterion provided by Charmaz (and Stol et al.) [11] [66].

Credibility: *Is there enough data to merit claims of the study?* – The synthesis of this thesis is based on the data set of two original field studies (Paper I and Paper II) and the observations of a third field study [53]. The data set is rich and constructed using a considerable degree of rigor. So, yes. There is enough data merit to our claims.

Originality: *Do the findings offer new insight?* – While the observation that *softer* issues in Software Engineering, in general, are not exhaustively reserached is general, we have observed a considerable research gap in regards to *cognitive load* in a software engineering context. The synthesis of Paper III offer novel *perspectives* on cognitive load in software engineering.

Usefulness: *Are the theories/models generated relevant for practitioners?* – While the use of *relevance to practitioners* as an evaluation criteria for models in Software Engineering is debated [65] [49], the findings of this thesis are general and describe problems and consequences of problems existing in the software engineering context. As such it is fair to say that the findings are relvant for practitioners.

Resonance: While writing the papers we omitted the concept of *resonance* as an evaluation criteria in PII/PIII – in hindsight this might have been a mistake, but at the time of writing it did not fully make sense. While we have not had the chance to evaluate the resonance of our findings in the populations studied, we note that we concluded the field work of PII with focus groups, thus adding to the rigor of the study.

10 Conclusion

While the research described this theseis has been explorative, from a phenomenological as well as from a methodological point of view, it has none the less provided relevant knowledge. In the papers we have started charting and and describing cognitive load in a software engineering context, and have proposed areas that, in all likelihood, would benefit from additional user support in terms of tooling.

We have further successfully deployed distributed cognition and cognitive load theory as scientific lenses for investigating and analysing cognitive load. Further we have (quite successfully) used grounded theory ethnography and grounded theory as methodology for qualitative inquiry in regard to cognitive load, using multi-method data collection and qualitative synthesis. We have also started to describe how a temporal dimension impacts software engineering, and we have found that a temporal synthesis is a considerable cause of cognitive load for people developing software in an agile and distributed fashion. The field studies have been equally challenging and rewarding, as has the methodological approach.

In conclusion, we have successfully used qualitative inquiry as a mean to shed light over cognitive load in software engineering. Despite the fact that *softer issues* has been recognized as important in software engineering for more than 50 years there is still an overwhelming emphasis on more technical aspects within the Software Engineering research community. Hence we see that there is a considerable amount of relevant future research in line with this thesis that can contribute to the body of knowledge within the research area.

11 Future research

Future research endeavours will in all likelihood include:

Further industrial case study/-ies to further explore the *perspectives* of cognitive load. One such study has been designed but postponed due to the limited availability of case companies during Covid-19. This will be redesigned and aligned with the findings of paper II & III, thus focusing more on version control, merge and branching. One aim would be to further establish a concrete understanding of what mechanisms cause cognitive load issues in a software engineering work context, and to see to what extent these differ from those identified when we studied novices. Such studies will be based on grounded theory and would include mixed method data set construction using interviews and observations. The observations would in all likelihood, on account of time constraints, be more focused on 'show & tell' (i.e. asking practitioners to demonstrate what they percieve as problematic). Such a case study will rely on the *perspectives* provided in Paper III as a lens for analysis.

While we did a rather limited literature survey on version control, branching and merging in Paper II, a more formal systematic mapping study of extant literature on version control and merge tools in software engineering seems warranted. In light of the findings of the limited literature review in Paper II, a more exhaustive mapping study will be conducted. This could possibly be extended outside of published papers, using *grey literature*.

Based on findings from Paper II a comparative benchmark of different IDEs and corresponding GIT integration, focusing on version control and merge operations and suggested design improvements is warranted. This could possibly be executed as one (or ideally several) Masters' thesis. It would of course be prudent to prototype and evaluate suggested design improvements. Such work would initially rely on Shneidermans *golden rules* [63] of user interface design and the *Perspectives* provided in Paper III as initial lenses of this type of research. Based on the findings of Paper III it makes sense to further investigate the temporal domain in software engineering. Ultimately this work could focus on bridging temporal cognitive gaps and develop suggestions for relevant mechanisms/designs for temporal tool support in SE context.

From a potential long term perspective (beyond the scope this research project), it would be interesting with a qualitative meta study on the less good aspects of agile software development and/or cognitive load as a consequence of agile methodologies, using grounded theory and meta-ethnograpy [15] [16]. This could possibly be followed up with a field study using grounded theory and would include mixed method data set construction using interviews and observations. Distributed cognition would, again, serve as a scientific lens – possibly in conjunction with cognitive load theory. In addition, from the findings of Paper III it would be interesting to further explore the concept of cognitive sustainability, or the long term consequenses of cognitive load and overload, in software engineering. Such a study would use grounded theory ethnography, but be focused more on interviews, rather than observation (observations as data collection method would in all likelihood be difficult in temporal terms). It would be interesting to deploy a constructivist approach in such as study, and it could also attept to capture additional aspects of *aging in software engineering*, as the ability to harbor cognitive load changes with age [72].

References

[1] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4):487–513, August 2011.

[2] Steve Adolph, Philippe Kruchten, and Wendy Hall. Reconciling perspectives: A grounded theory of how people manage the process of software development. *Journal of Systems and Software*, 85(6):1269–1286, June 2012.

[3] Mats Alvesson and Kaj Sköldberg. *Reflexive Methodology - New Vistas for Qualitative Research*. SAGE Publications, London, UK, 3rd edition, 2018.

[4] Yuichiro Anzai and Herbert A. Simon. The Theory of Learning by Doing. *Psychological Review*, 86(2):124–140, 1979.

[5] Sebastian Baltes and Stephan Diehl. Towards a theory of software development expertise. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 187–200, Lake Buena Vista, FL, USA, 2018. ACM Press.

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

[7] Alan F. Blackwell, Marian Petre, and Luke Church. Fifty years of the psychology of programming. *Int. J. Hum. Comput. Stud.*, 131:52–63, 2019.

[8] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, January 2006.

[9] Jim Buchan, Didar Zowghi, and Muneera Bano. Applying Distributed Cognition Theory to Agile Requirements Engineering. In Nazim Madhavji, Liliana Pasquale, Alessio Ferrari, and Stefania Gnesi, editors, *Requirements Engineering: Foundation for Software Quality*, Lecture Notes in Computer Science, pages 186–202, Cham, 2020. Springer International Publishing.

[10] Stuart Card, Jock Mackinlay, and Ben Shneiderman. *Readings in Information Visualization: Using Vision To Think*. January 1999.

[11] Kathy Charmaz. *Constructing Grounded Theory*. SAGE Publications, London, UK, 2nd edition, 2014.

[12] Kathy Charmaz and Ricard Mitchell. Grounded Theory in Ethnography. In *Handbook of Ethnography*. SAGE Publications, London, UK, 2001.

[13] Michelene T. H. Chi, Robert Glaser, and Ernest Rees. Expertise in Problem Solving. Technical Report TR-5, Pittsburg Univ PA Learning Research and Development Center, May 1981.

[14] Luke Church, Emma Soderberg, and Elayabharath Elango. A case of computational thinking: The subtle effect of hidden dependencies on the user experience of version control. In Benedict du Boulay and Judith Good, editors, *Proceedings of Psychology of Programming Interest Group Annual Conference*, pages 123–128, Brighton, United Kingdom, June 2014.

[15] Daniela S. Cruzes and Tore Dybå. Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 275–284, September 2011. ISSN: 1949-3789.

[16] Daniela S. Cruzes and Tore Dybå. Research synthesis in software engineering: A tertiary study. *Information and Software Technology*, 53(5):440–455, May 2011.

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

[18] Santiago Perez De Rosso and Daniel Jackson. Purposes, Concepts, Misfits, and a Redesign of Git. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 292–310, New York, NY, USA, 2016. ACM.

[19] Nicolas Debue and Cécile van de Leemput. What does germane load mean? An empirical contribution to the cognitive load theory. *Frontiers in Psychology*, 5, 2014.

[20] David Fetterman. *Ethnography: step-by-step*. SAGE Publications, Thousand Oaks, California, USA, 2010.

[21] Nick V. Flor and Edwin L. Hutchins. Analyzing distributed cognition in software teams: a case study of team programming during perfective maintenance. In Jürgen Koenemann-Belliveau, Thomas Glenn Moher, and Scott P. Robertson, editors, *Proceedings of Empirical Studies of Programmers*, pages 36–64, Norwood, NJ, USA, 1991. Ablex Publishing Corporation.

[22] Thomas Fritz and Sebastian C. Müller. Leveraging Biometric Data to Boost Software Developer Productivity. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 66–77, March 2016.

[23] Barney G. Glaser. *Theoretical Sensitivity*. Sociology Press, CA, USA, 1978.

[24] Barney G. Glaser. *Emergence vs Forcing - Basics of Grounded Theory Analysis*. Sociology Press, CA, USA, 1992.

[25] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory*. AldineTransaction, New Jersey, USA, 1967.

[26] Lucian Gonçales, Kleinner Farias, Bruno da Silva, and Jonathan Fessler. Measuring the Cognitive Load of Software Developers: A Systematic Mapping Study. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 42–52, May 2019. ISSN: 2643-7171.

[27] Jan Gulliksen, Ann Lantz, Åke Walldius, Bengt Sandblad, and Carl Åborg. Digital arbetsmiljö, en kartläggning (RAP 2015:17). Technical report, 2015.

[28] Sean Hansen and Kalle Lyytinen. Distributed Cognition in the Management of Design Requirementsdistributed cognition in the management of design requirements. In Robert C. Nickerson and Ramesh Sharda, editors, *Proceedings of the 15th Americas Conference on Information Systems*, page 266, San Francisco, California, USA, August 2009. Association for Information Systems.

[29] Rashina Hoda, James Noble, and Stuart Marshall. Grounded theory for geeks. In *Proceedings of the 18th Conference on Pattern Languages of Programs*, PLoP '11, pages 1–17, Portland, Oregon, USA, October 2011. Association for Computing Machinery.

[30] Rashina Hoda, James Noble, and Stuart Marshall. The impact of inadequate customer collaboration on self-organizing Agile teams. *Information and Software Technology*, 53(5):521–534, May 2011.

[31] Rashina Hoda, James Noble, and Stuart Marshall. Self-Organizing Roles on Agile Software Development Teams. *IEEE Transactions on Software Engineering*, 39(3):422–444, March 2013. Conference Name: IEEE Transactions on Software Engineering.

[32] James Hollan, Edwin Hutchins, and David Kirsh. Distributed Cognition: Toward a New Foundation for Human-computer Interaction Research. *ACM Trans. Comput.-Hum. Interact.*, 7(2):174–196, June 2000.

[33] Nina Hollender, Cristian Hofmann, Michael Deneke, and Bernhard Schmitz. Integrating cognitive load theory and concepts of human–computer interaction. *Computers in Human Behavior*, 26(6):1278–1288, November 2010.

[34] Edwin Hutchins. *Cognition in the Wild*. MIT Press, 1995.

[35] David Kirsh. A Few Thoughts on Cognitive Overload. *Intellectia*, (30):19–51, 2000.

[36] Richard J. Koubek, Gavriel Salvendy, Hubert E. Dunsmore, and William K. LeBold. Cognitive issues in the process of software development: review and reappraisal. *International Journal of Man-Machine Studies*, 30(2):171–191, February 1989.

[37] K.Ronald Laughery and Kenneth R. Laughery. Human factors in software engineering: A review of the literature. *Journal of Systems and Software*, 5(1):3 – 14, 1985.

[38] Per Lenberg, Robert Feldt, and Lars Göran Wallgren. Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, 107:15–37, September 2015.

[39] George Mangalaraj, Sridhar Nerur, RadhaKanta Mahapatra, and Kenneth H. Price. Distributed Cognition in Software Design: An Experimental Investigation of the Role of Design Patterns and Collaboration. *MIS Quarterly*, 38(1):249–A5, March 2014.

[40] Vivian Martin. Using Populat and Academic Literature as Data for Formal Grounded Theory. In *The SAGE Handbook of Current Developments in Grounded Theory*. SAGE Publications, London, UK, 2019.

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

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

[43] Peter Naur and Brian Randell. Software engineering: Report on a conference sponsored by the nato science committee. Technical report, Scientific Affairs Division, NATO, January 1969.

[44] Ikujiro Nonaka, Ryoko Toyama, and Noboru Konno. SECI, Ba and Leadership: a Unified Model of Dynamic Knowledge Creation. *Long Range Planning*, 33(1):5–34, February 2000.

[45] Santiago Perez De Rosso and Daniel Jackson. What's Wrong with Git?: A Conceptual Design Analysis. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 37–52, New York, NY, USA, 2013. ACM.

[46] Narayan Ramasubbu, Chris F. Kemerer, and Jeff Hong. Structural Complexity and Programmer Team Strategy: An Experimental Test. *IEEE Transactions on Software Engineering*, 38(5):1054–1068, September 2012.

[47] Hugh Robinson, Judith Segal, and Helen Sharp. Ethnographically-informed empirical studies of software practice. *Information and Software Technology*, 49(6):540–551, June 2007.

[48] Colin Robson. *Real World Research*. Malden: Blackwell, 2nd edition, 2002.

[49] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, March 2012.

[50] Caitlin Sadowski and Thomas Zimmermann, editors. *Rethinking Productivity in Software Engineering*. Apress, Berkeley, CA, 2019.

[51] Todd Sedano, Paul Ralph, and Cecile Péraire. Sustainable Software Development through Overlapping Pair Rotation. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '16*, pages 1–10, Ciudad Real, Spain, 2016. ACM Press.

[52] Todd Sedano, Paul Ralph, and Cecile Péraire. Lessons Learned from an Extended Participant Observation Grounded Theory Study. In *2017 IEEE/ACM 5th International Workshop on Conducting Empirical Studies in Industry (CESI)*, pages 9–15, May 2017.

[53] Todd Sedano, Paul Ralph, and Cecile Péraire. Software Development Waste. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 130–140, May 2017.

[54] Todd Sedano, Paul Ralph, and Cecile Péraire. The Product Backlog. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 200–211, May 2019. ISSN: 1558-1225.

[55] Helen Sharp, Yvonne Dittrich, and Cleidson R. B. de Souza. The Role of Ethnographic Studies in Empirical Software Engineering. *IEEE Transactions on Software Engineering*, 42(8):786–804, August 2016.

[56] Helen Sharp, Rosalba Giuffrida, and Grigori Melnik. Information Flow within a Dispersed Agile Team: A Distributed Cognition Perspective. In *Agile Processes in Software Engineering and Extreme Programming*, Lecture Notes in Business Information Processing, pages 62–76. Springer, Berlin, Heidelberg, May 2012.

[57] Helen Sharp and Hugh Robinson. An Ethnographic Study of XP Practice. *Empirical Software Engineering*, 9(4):353–375, December 2004.

[58] Helen Sharp and Hugh Robinson. A Distributed Cognition Account of Mature XP Teams. In *Extreme Programming and Agile Processes in Software*

Engineering, Lecture Notes in Computer Science, pages 1–10. Springer, Berlin, Heidelberg, June 2006.

[59] Helen Sharp and Hugh Robinson. Collaboration and co-ordination in mature eXtreme programming teams. *International Journal of Human-Computer Studies*, 66(7):506–518, July 2008.

[60] Helen Sharp, Hugh Robinson, and Marian Petre. The role of physical artefacts in agile software development: Two complementary perspectives. *Interacting with Computers*, 21(1-2):108–116, January 2009.

[61] Helen Sharp, Hugh Robinson, Judith. Segal, and Dominic Furniss. The role of story cards and the wall in XP teams: a distributed cognition perspective. In *AGILE 2006 (AGILE'06)*, pages 11 pp.–75, July 2006.

[62] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15–24, November 1990.

[63] Ben Shneiderman, Catherine Plaisant, Maxine Cohen, Steven Jacobs, Niklas Elmqvist, and Nicholas Diakopoulos. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson, 6th edition, 2016.

[64] Janet Siegmund. Program Comprehension: Past, Present, and Future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 13–20, March 2016.

[65] Klaas-Jan Stol and Brian Fitzgerald. Theory-oriented software engineering. *Science of Computer Programming*, 101:79–98, April 2015.

[66] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 120–131, May 2016.

[67] Margaret-Anne Storey, Neil A. Ernst, Courtney Williams, and Eirini Kalliamvakou. The who, what, how of software engineering research: a socio-technical framework. *Empirical Software Engineering*, 25(5):4097–4129, September 2020.

[68] John Sweller and Paul Chandler. Why Some Material Is Difficult to Learn. *Cognition and Instruction*, 12(3):185–233, September 1994.

[69] Edward R Sykes. Interruptions in the workplace: A case study to reduce their effects. *International Journal of Information Management*, page 10, 2011.

[70] Andrew Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. PhD thesis, School of Computing Sciecne, Simon Fraser University, 2002.

[71] Christopher D. Wickens, Justin G. Hollands, Simon Banbury, and Raja Parasuraman. *Engineering Psychology & Human Performance*. Psychology Press, August 2015.

[72] Arthur Wingfield, Elizabeth A. L. Stine, Cindy J. Lahar, and John S. Aberdeen. Does the capacity of working memory change with age? *Experimental Aging Research*, 14(2):103–107, June 1988.

[73] Luciana A. M. Zaina, Helen Sharp, and Leonor Barroca. UX information in the daily work of an agile team: A distributed cognition analysis. *International Journal of Human-Computer Studies*, 147, March 2021.

Included papers

CHAPTER I

COGNITIVE LOAD DRIVERS IN LARGE SCALE SOFTWARE DEVELOPMENT

Citation:
Daniel Helgesson, Emelie Engström, Per Runeson, and Elizabeth Bjarnason. Cognitive Load Drivers in Large Scale Software Development. In Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE '19, pages 91-94, Piscataway, NJ, USA, 2019. IEEE Press.

Abstract

Software engineers handle a lot of information in their daily work. We explore how software engineers interact with information management systems/tools, and to what extent these systems expose users to increased cognitive load. We reviewed the literature of cognitive aspects, relevant for software engineering, and performed an exploratory case study on how software engineers perceive information systems. Data was collected through five semistructured interviews. We present empirical evidence of the presence of cognitive load drivers, as a consequence of tool use in large scale software engineering.

1 Introduction

Software engineering is a socio-technical endeavour where the technical side of the phenomena seems to be more studied than the social side [5], and as a consequence knowledge of a cognitive/ergonomic perspective of software development, and the tools associated with these activities, appears rather small. Further, we see no clear indications of a significant impression on the software engineering community in terms of understanding the cognitive work environment of software engineers [13] [12].

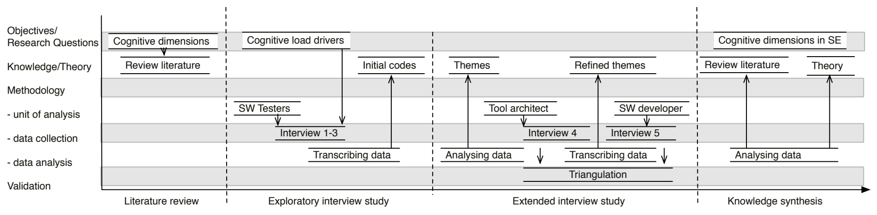

Figure 1: A description of how the case study evolved in terms of objectives, knowledge and research activities

In a 2002 dissertation, Walenstein observed that there is a need for cognitive reasoning in the design process of software development tools, and further that there has been little research done in the area [13], a claim largely substantiated by Lenberg et al. [5].

More recently, in a 2015 report 'Digital Work Environment', Gulliksen et al. made an effort to analyse the societal consequences of large-scale digitalisation of human labour, in general [3]. In the report the authors present a literature survey, providing updated insight into the research area. The survey found only 36 relevant articles. In addition, the authors also present a taxonomy of 'Cognitive work environment problems'.

In this study we aim to explore, and establish, a broader understanding of the cognitive work environment of software engineers and the cognitive dimensions of the tools used. Specifically, we aim to explore *cognitive load*, induced on users by information systems or tools. We present results from an exploratory industrial case study based on thematic analysis of interviews, as well as a literature overview. Our contribution lies in presenting in vivo observation of cognitive problems associated with tool use in large-scale software engineering.

2 Research questions

The purpose of this study is to gain insight into the problem domain of cognitive load, primarily as a consequence of tool use, in large scale software development. Hence it is exploratory in nature, and focuses on two tools central for communication and knowledge management at the case company.

The overall exploratory purpose is refined into two research questions:

RQ1 Which types of cognitive load drivers can be observed in large-scale software engineering, primarily as a consequence of tool use?

RQ2 How do software engineers perceive the identified cognitive load drivers in their digital work environment?

The research questions are anchored in software engineering and cognitive science literature, and addressed by interviewing practitioners. The first question uses the cognitive literature as a lens, while presenting empirical observations from the interview material. The second question reports the interviewees' perception of problems found in RQ1.

3 Method

We conducted a four stage case study, using a flexible design [10]; consisting of *literature review, interview study, extended interview study*, and *knowledge synthesis*. To mature the knowledge, we iterated reviewing literature, conducting interviews, transcribing and analysing data.

The case company is an international corporation, the studied division develops consumer products in the Android ecosystem. The software is embedded in handheld devices. The studied development site of the company has 1000 employees and developers work in cross-functional teams using an agile development process. The development environment is primarily based on the toolchain associated with the Android ecosystem.

The case study is informed by five interviews. We started with three interviews of people from a test team – a test manager (i1), and two testers (i2, i3). After an initial analysis of the of interviews, one additional interview (a tool architect, i4) was conducted to to provide background. A final interview (a software developer, i5) was added to validate the findings. All interviews were recorded, transcribed by the first author, then iteratively coded and analysed by the authors collectively.

The first three interviews were semistructured, following an interview guide. These three interviews were conducted by the first and fourth authors collectively. The fourth interview was specifically centered around the two most discussed tools of interviews 1-3. The main architect of these tools was selected for this interview, due to being able to provide background on the development history, as well as rationales on certain design decisions. This interview was conducted by the first author. The fifth person interviewed was selected as he had worked with the defect management system since its introduction ten years ago. This interview followed the protocol of the first three. The interview was conducted by the first author.

The data was analysed as outlined in A set of initial codes were identified during the transcription of the first three interviews by the first author. These codes were then used to create a set of themes in two iterations. The fourth and fifth interviews were used to validate and extend the initial set of interviews. The themes were then refined, and reapplied to all five interviews to extract information on cognitive load drivers. The main iterations of the thematic analysis were executed by three authors collectively.

The classification of cognitive load drivers was then validated against the classification presented by Gulliksen et al.

4 Literature overview and analysis

4.1 Purpose and strategy

The purpose of our literature overview is twofold as we aim to: i) provide an overview of cognitive research, relevant and related to software engineering; and ii) present qualitative observations from relevant cognitive literature, not specifically targeting software engineering, that can serve as a step stone for further research. It should be noted that we, by no means, claim that this is a formal, nor systematic, literature study [6]. What we present are, essentially, *qualitative* findings from an *exploratory* literature study, executed as a part of an *exploratory* case study.

4.2 Exploratory survey of cognitive research related to Software Engineering

In her keynote at ASE 2018 [8], Murphy presented an updated example of cross industrial/academic software engineering research bordering on cognition, which emphasizes the relevance of context in software engineering. Arguably, the most researched cognitive aspect of software engineering, historically, is program comprehension – a comprehensive overview of past, current and future research directions of program comprehension is presented by Siegmund [12], while Sharafi et al. systematically reviewed software engineering research, that use eye-tracking [11]. Human aspects of software engineering was studied by Lenberg et al. [5], claiming that less than 5% of the articles studied "a 'soft' or human-related topic".

While there is a lot of research on Human Computer Interaction in general, very little is specifically looking at software development and tools. However, there are examples of articles on usability of software tools, e.g., Myers et al. [9], Dillon and Thomas [2]; as well as on design of software tools, e.g., Holzinger [4].

In conclusion, the problem with cognitive research in Software Engineering appear as being twofold i) not very much research has been done beyond code comprehension, and ii) the research has rarely been executed in the real world, 'in vivo', context.

4.3 Practical implications of human cognitive limitations

It has, since Miller's 'The Magical Number Seven, Plus or Minus Two' [7] from the 1950's, been generally accepted that the human working memory is finite. Accepting that 'the capacity of the human working memory' and cognitive bandwidth (i.e. the amount of cognitive load a human mind can process) are closely related, it can be deduced that unnecessary cognitive, or mental, load is likely to decrease the cognitive bandwidth, which over time translates into 'throughput' or 'work'. Miller's findings are generally accepted, although later research shows the actual bandwidth being lower than Miller proposed [1]. Herein lies the main rationale for studying cognitive load drivers - if all tools (or tasks) induce cognitive load on the subject, keeping the undesired cognitive load to a minimum will allow for more of the cognitive bandwidth being used for relevant chores and not practical waste.

5 Results

We analysed the interview data, partially in the light of the general cognition literature summarised above, and identified different types of load drivers (RQ1), and perceived problems in the digital work environment (RQ2).

5.1 Identified cognitive load drivers (RQ1)

During the analysis of the interview material, we identified factors mentioned by interviewees, which were interpreted as having an impact on the cognitive load of the developers, i.e. being *cognitive load drivers*. Then, in the coding, these factors were grouped into *items*, which characterise the cognitive load, e.g. lack of adaptation of a tool to the work task. The items were then grouped into *themes* and finally clustered into the object of the cognitive load drivers.

The two main clusters of load drivers in the thematic analysis are: *Tool* – cognitive load directly associated to use of tools; and *Information* – cognitive load associated to information management, information flow and information load respectively. In addition we found a third cluster: *Work process* – cognitive load derived from processes and organisational aspects of the workplace.

Tool

The aspects of cognitive load associated to *tools*, are of three kinds, namely *intrinsic, delay* related, and *interaction*.

Intrinsic aspects include lack of functionality, manifesting itself as use of tools with poor *adaptability/suitability* to the purpose for which they were intended, including lack of *functionality*. For example, the search functionality is not up to date, which makes it hard to find old defect reports. Further, *lack of stability or reliability* prevent users from trusting the tools. '...if it crashes all data is lost' (i2). *Overlap* and *lack of integration* prevent users from working efficiently, as it causes redundant work (e.g. copying information from one tool/system to another). Finally, *comprehension*, i.e. understanding what is going on when using the tool, is a cognitive load factor – 'It is not obvious how it should be used.' (i3).

Delay is an absence of response in a *tool* or system. It can be observed in our interviews as a state of forced concentration when the user is forcing information to remain in working memory while waiting for the tool to respond; and it can be observed in terms of system downtime. We observed cognitive load from *response* delays at the micro level, as noted by interviewee i3: 'But when it is non responsive you loose focus. You can't just stay focused.' Further, *downtime* on the macro level scale beyond a few seconds effectively prevents all work with the system. 'It has happened that [the tool] has been down for longer periods, especially after upgrades' (i2).

Three different aspects of cognitive loads were identified which considered consequences of *interaction* issues with *tools*. *Unintuitive* implies that the users find the tools hard to use, causing frustration and unnecessary cognitive load to the user as he/she must repeatedly find out how to complete a task. 'I felt that I had to transform my [mental] model to some kind of database model, in order to understand how the tool worked' (i5). *Inconsistent* systems or different parts of a system work rather differently in terms of interaction, causing frustration to the user as he/she repeatedly must determine how to solve a task, in the current context, specifically. *Cumbersome* interaction is when functionality is missing or implemented in such a way that the user is forced to waste energy doing what is considered unnecessary work.

Information

The quality of the *information* in itself is the second cluster of cognitive load drivers. The different aspects of cognitive load associated to the *integrity* of information are related to *incompleteness* of information, which causes the user to spend effort in asserting that information is complete. The lack of *reliability* of the information is causing the user to spend effort in asserting that information is correct and up-to-date (e.g. caused by lack of version control). Interviewee i2 stated: 'The main issue is that there is no revision handling/version control'. Finally, the *temporal traceability* of information over time is needed to help a user to bridge a temporal gap in order to assess the current situation, e.g. see if an issue has been reported before, or if an error has been fixed in an earlier release.

The different aspects of cognitive load associated with the *organisation* of *information* is related to *location*, i.e. having a hard time finding information, *retrieval*, i.e. having a hard time retrieving information, *distribution*, i.e. not knowing where to store information or whom to distribute it to, *overview/zoom*, i.e. absence of overview or zoom views cause cognitive load when browsing information. 'There is no overview' (i2). Finally, *structure* is when the information is structured or organised in a cumbersome fashion, e.g. as mentioned by i3, when test results are not saved per test run, something that later cause problems with information accessibility and affects the users' ability to find correct information. The same was indicated by i1, who stated that: 'If I rerun this test project I only see the last result', indicating that the visualisation delta between the current run and the previous run would be beneficial.

Work process

The aspects of cognitive load associated to *work process* mainly gravitate around lack of support, manifesting itself in wasted effort; either by doing unnecessary work, or by having to spend effort in finding out how to solve a task in a specific tool or in a certain team. Cognitive load drivers in this cluster are related to *lack of automation*, *wasted effort* (be it unnecessary or redundant), illustrated by one interviewee (i2), stating on mandatory information fields to be filled out even when not needed. Further, *ad hoc* implementation of tools or processes, lead to wasted or inconsistent work process, since they are implemented differently in different parts of the organisation. Finally, *lack of understanding* of the intrinsics of a large organisation can be a load driver itself.

Validation of empirical findings

We validated our collection of cognitive load drivers against the set of 'Cognitive work environment problems, identified by Gulliksen et al. [3]. In light of that they studied digitalization of work in general, it is quite natural that the cognitive issues identified differ somewhat compared to our findings. That being said, there does not appear to be any contradictions between the two sets, and while not identical, they are largely similar.

5.2 Perceived problems (RQ2)

The main issue with the test management tool, was missing revision control and absence of revision history. Furthermore, all interviewees had noted that there was no strict ownership, meaning that any user could change test cases and test scripts as they saw fit. The main issue with the defect management tool, apart from cumbersome interaction, was that omission of search functionality - which makes it very hard to find error reports (e.g. duplicated, closed or obsolete error reports). To exemplify, one user used the notification e-mails supplied by the defect management tool to create his own temporal model/history of the error reports handled by his team (i5). Furthermore, as a consequence of the interaction issues with the tool, the quality of the data extracted from the system was perceived to be quite unreliable (i4). The same interview revealed an interesting observation regarding the selection/acquisition of systems. Despite the discontent among the users, and the apparent flaws of the tool, these aspects were not considered part of the business case when the defect management system was replaced. Instead, the sole rationale for the change was, according to the interviewee, that the licensing cost of the new tool/system was significantly lower than that of the old tool.

5.3 Summary

We conclude that cognitive load drivers are indeed present and have considerable impact in large-scale software engineering, in this specific case, and that the load drivers identified gravitate around three clusters; 'Tools', 'Information' and 'Work process'.

6 Limitations

The literature review is not complete, in terms of covering all cognitive science literature relevant for software engineering, nor all software engineering literature related to cognition. The goal of the literature review and analysis is to create a foundation for the exploratory case study. Thus it is sufficient to have relevant cognitive science literature as a basis, which we ensure by having some core scholars represented, such as Miller, [7] and Siegmund [12]. In regards of software engineering literature, we observe that there are some aspects of cognition studied, which confirms it being a relevant perspective.

The case study is conducted at one company, with a limited sample size of interviews, limiting its external validity. However, the aim of the study is to explore the field. We have, in all likelihood, missed some cognitive load drivers, that are not present in the case study context. Our validation against Gulliksen et al's taxonomy [3] shows a sound mapping of the cognitive load drivers in a general working context to those found in this study. The authors have long working and collaboration history with the case company, which helps improving the construct validity although it may introduce bias. However, as we only provide evidence of the presence of cognitive load drivers, that does not limit the validity of the findings. We conclude, that with the exception of research on program comprehension, little is published on cognitive aspects of tool use in software engineering. We also observe that there is indeed work in the cognitive science area relevant for software engineering, which could be used to lay out the theoretical basis for studying and improving software engineering from a cognition point of view. Grounded in the literature in the field, we conclude that we in this exploratory study have found, and presented, empirical evidence of the presence of cognitive load can be observed in large-scale software engineering, RQ1. We further conclude that it is indeed is a problem for practitioners, RQ2. Our classification of the load drivers found gravitates, or clusters, around 'Information', 'Tools' and 'Work/Process'. This research was financed by projects EASE and ELLIT respectively.

References

[1] Nelson Cowan. The Magical Mystery Four: How Is Working Memory Capacity Limited, and Why? *Current Directions in Psychological Science*, 19(1):51–57, February 2010.

[2] Brian Dillon and Richard Thompson. Software development and tool usability. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4, May 2016.

[3] Jan Gulliksen, Ann Lantz, Åke Walldius, Bengt Sandblad, and Carl Åborg. Digital arbetsmiljö, en kartläggning (RAP 2015:17). Technical report, 2015.

[4] Andreas Holzinger. Usability Engineering Methods for Software Developers. *Commun. ACM*, 48(1):71–74, January 2005.

[5] Per Lenberg, Robert Feldt, and Lars Göran Wallgren. Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, 107:15–37, September 2015.

[6] Yair Levy and Timothy J. Ellis. A Systems Approach to Conduct an Effective Literature Review in Support of Information Systems Research. *Informing Science: The International Journal of an Emerging Transdiscipline*, 9:181–212, 2006.

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

[8] Gail C. Murphy. The Need for Context in Software Engineering (IEEE CS Harlan Mills Award Keynote). In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 5–5, New York, NY, USA, 2018. ACM.

[9] Brad A. Myers, Aandrew J. Ko, Thomas D. LaToza, and YoungSeok Yoon. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer*, 49(7):44–52, July 2016.

[10] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. *Case study research in software engineering: guidelines and examples*. Wiley, 2012.

[11] Zohreh Sharafi, Zéphyrin Soh, and Yann-Gaël Guéhéneuc. A systematic literature review on the usage of eye-tracking in software engineering. *Information and Software Technology*, 67:79–107, November 2015.

[12] Janet Siegmund. Program Comprehension: Past, Present, and Future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 13–20, March 2016.

[13] Andrew Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. 2002.

CHAPTER II

A GROUNDED THEORY OF COGNITIVE LOAD DRIVERS IN AGILE SOFTWARE DEVELOPMENT

Citation:
– working manuscript

Abstract

Context: Agile software development in teams is a socio-technical iterative activity involving a network of people, software and hardware. The capacity of the human working memory is limited, and cognitive load is induced on developers by tasks, software systems and tools.

Objective: The purpose of this paper is to identify the largest cognitive challenges faced by novices developing software in teams.

Method: Using Grounded Theory, we conducted an ethnographic study for two months following four ten person novice teams, consisting of computer engineering students, developing software systems.

Result: This paper identifies version control and merge operations as the largest challenge faced by the novices. The literature studies reveal that little research appears to have been carried out in the area of version control from a user perspective.

Limitations: A qualitative study on students is not applicable in all contexts, but the result is credible and grounded in data and substantiated by extant literature.

Conclusion: We conclude that our findings motivate further research on cognitive perspectives to guide improvement of software engineering and its tools.

1 Introduction

With the advent of agile methodologies, the socio-technical character of software engineering has been further promoted. With its emphasis on 'individuals and interactions *over* processes and tools'[1], the Agile manifesto makes a strong promotion of the social side. The technical side of the phenomena seems to be more studied than the social side [22] [39], although human factors were part of software engineering already from its inception. For example, *personell factors* were discussed at the NATO conference 1968 [25]. In their summary of 50 years of the psychology of programming, Blackwell et al. synthesize research in the intersection between several research communities [3]. Referring to Bødker [4], they see three major waves of research, namely '*in the first wave, cognitive psychology and human factors; in the second wave, social interaction within work settings; and in the third wave, a focus on everyday life and culture.*'.

As a contrast to the third wave scaling down towards the craft or hobby programming, we would like to bring the up-scaling of software engineering into attention. The complexity added by composing systems of systems, the increasing and constant change enabled by agile methodologies and DevOps, the more and more powerful tools for software engineering – all add to the cognitive load of developers and managers of software engineering projects. Blackwell et al. claim in their proposal for a future research agenda, that '*'Design thinking', solving 'wicked' problems, and reasoning more broadly about software systems and systems-of-systems has not received the sort of attention that has been devoted to, say, program comprehension.*' [3].

In an attempt to more broadly adress the cognitive load induced from these trends, we studied *cognitive load drivers* in large scale software development [16] and found three clusters of drivers, namely *tools*, *information*, and *work & process*. We also noticed that the temporal perspective of software development, particularly revision control created specific problems.

To further advance the understanding of cognitive load in software engineering, we set out to ethnographically study agile software engineering project, using grounded theory. As we hypothesise that some of the cognitive leads are compensated and mitigated through increased experience and work-arounds learned over the years, we choose to study novice software engineers [20]. Our study context is quite advanced for novices, an agile software engineering course, running for 14 weeks, in which students work in 10 person teams in a simulated work environment, adhering to XP principles. We observe four teams out of a total of twelve teams participating in the course.

Our research goals are to identify the most dominant *cognitive load drivers* and to observe differences and similarities between groups with different characteristics. We combine the teacher role of on-site customer with the ethnographer role, taking field notes of the observations. Further, we collect weekly questionnaires,

[1] `https://agilemanifesto.org`

short reflection notes from the students, and arrange a focus group discussion with each team. We use grounded theory practices in coding all the material, from which our theory emerges.

We conclude that version control, branching and merge operations is the dominant load factor in the projects, and thus we explore this phenomenon in detail.

2 Method

2.1 Grounded theory

Grounded theory (GT) is a systematic and rigorous methodological approach for inductively generating theory from data [14] [6] [37]. Stemming from social sciences, GT was developed by sociologists Glaser and Strauss, as a qualitative inductive reaction to the quantitative hypotetico-deductive reserach paradigms dominant in the 1960's. The main difference, apart from being qualititative rather than quantitative, is that the purpose of GT aims at generating theory, rather than to be used as an instrument for validation, or testing, of theory [37]. It is iterative and explorative [6] in nature, and thus suitable for answering open ended questions such as *what's going on here?* [37] [1]. Specific guidelines and suggestions for GT research in Software Engineering are provided by Stol et al. [37].

We primarily opted for Charmaz GT handbook [6] as guidelines (in addition we also consulted earlier works by Glaser [12] [13]), specifically using grounded theory ethnography [7] – an approach that gives '*priority to the studied phenomenon or process – rather than the setting itself*' [6]. The ethnographic approach allows for exploring not only *what* practitioners do, but also *why* they do it [30]. Core elements in the ethnographic approach is the empathic approach emphto describe another culture from the members point of view and the intrinsic *analytical stance* [30]. As with grounded theory, modern ethnography also stems from social sciences [30]. Not extensively used in Software Engineering [30], it has however been used to study agile teams [33] [32].

2.2 Research goals

Central to 'original' Glaserian GT and Charmaz Constructivist GT is that the actual/final research questions are not defined up front. In the case of Glaser the researcher should start with an *area of interest* [13] [37], in the case of Charmaz the researcher should start with *initial research questions* that evolve through the study [6] [37]. We decided to pursue two open ended research goals:

A) To identify the most dominant *cognitive load drivers* from the *novice point of view*, and

B) To chart what differences or similarities that can be observed between the different group compositions.

2.3 Case description

The course that we used as study object is a mandatory course for sophomore computer science students aiming at teaching practical software development in teams using agile methodology. The course runs for two terms (14 weeks) and consists of one study block (seven weeks) consisting of lectures and practical lab work, and one study block (seven weeks) in which the students work together as 10 person teams, largely adhering to XP principles [32] developing a software product. All teams develop a software system based on the same basic stories, but the stories are somewhat open ended leaving room for differentiation. The teams are coached by two senior students undertaking a course in practical software coaching, that runs in parallel for the same duration. PhD students serve as *customer* for 3-4 teams each.

The teams develop their system for a term (seven weeks) in 6 full day sprints, each preceded by a two hour planning session in which the *cost/effort* for the user stories are estimated by the students and prioritised by the *customer*. The students make 3–4 incremental releases during the project, roughly with a cadence of one release every two sprints.

2.4 Design considerations

We opted for a flexible case study design [29], to allow for improvisation based on observations and forces outside of our control (which once you take research into the wild are plentiful). Once in the field, flexibility becomes utterly important [30] as the researcher must be ready to adapt to changing situations quickly.

We had a strict time box for our field study, since the course executed over the duration seven weeks with one day sprints on Mondays, following a two hour planning session on the Wednesday before. Apart from the fixed schedule for observations we also had to take into account the work load of the students when injecting experiments and eliciting interviews. We had the ambition to cause as little disturbance as possible. In order to achieve triangulation we opted to collect as many data sources as possible.

We also decided to use Distributed Cognition [18] as initial lens for our observations. While the use of an initial lens could be thought of by some readers as contradictory to the central tenet in grounded theory, we hold this (potential) critique as moot. We were targeting observations of a cognitive load drivers in interconnected network of people and digital tools, so we needed some starting point for our observations.

2.5 Student selection

Firstly we anonymously picked 14 student candidates, based on a high grade (grade average in excess of 4.5 on five grade scale, where *pass* is denoted as

3) in the first two programming courses, and a lower grade (i.e. *pass* or *incomplete/fail*) grade in multidimensional calculus. Secondly we anonymously picked 14 student candidates based on a high grade (grade higher than *pass* on five grade scale, where *pass* is denoted as 3) in multidimensional calculus, regardless of their grade in programming courses.

The two anonymous candidate lists were then sent to the course responsible who then created one experimental group each out of the two candidate lists and two randomly selected groups. After this process we had four groups in total. It should be noted that the authors at no point in time were informed of what group consisted of what selection.

2.6 Consent

Together the course responsible and the first author ultimately reached the conclusion that the optimal solution (in regards to time constraints and complexity) was to inform the students in the four groups at the start of the course that we would be carrying out research throughout the course, describe the overall purpose/general research goal of the study, that we were looking at the groups and not the individual members and offer any student not willing to participate to change groups prior to the first sprint. No student asked to exchange groups.

In every interaction that was recorded or photographed, we actively asked every student participating for permission, while pointing out that everything expressed in the exchange would be anonymous and confidential, and that no recordings would be distributed outside of the three researchers participating in the study.

2.7 Data collection

The first and the second author followed all planning sessions in parallel. As we had to monitor sessions in parallel we opted to alternate between observing in pairs and by ourselves. All in all we covered 24 planning sessions where the first author actively participated in the meetings acting as *customer on site* providing students with clarifications of stories, priorities etc, while the second author passively observed. After each session we spent, roughly, 15 minutes discussing what we had observed. Field notes were written by hand, and after the termination of the field work compressed in *memo* form. The first author actively participated in all full day sprints while acting as *customer on site*. The four teams were situated in two computer labs, allowing for observation of two teams simultaneously. Field notes were written by hand, and after the termination of the field work compressed in *memo* form.

In addition, we added a weekly questionnaire to be filled out by each student after every sprint (all in all 4*10*6 questionnaires) in order to follow up on what we had observed so far throughout the project. The first two weeks the questionnaire targeted sources of information and information tools used by the students. In the third and fourth questionnaire we introduced check boxes and free text space, allowing the students to express what they perceives as the major problems they had been challenged by throughout the project. In the fifth questionnaires questions were added to capture the outcome of experiment 1. The final questionnaire was extended with questions regarding team spirit and over all satisfaction. The aggregated response rate for all 24 sets of questionnaires (6 for each team) were 93% (out of the 240 questionnaires we handed out we got 223 in return, and no single set had a lower response rate than 8/10).

Further, as a requirement of the course all students wrote short individual reflections after each sprint, as a retrospect exercise. After the course we aggregated these pages, anonymised all content and created one .csv file per team with the content broken down in line-by-line format for *open coding*.

After the final sprint we held one hour long focus group discussion with each team. The discussions took place in two by two parallel sessions, Two instances were held by the first author, one by the second and third author collectively and one by the second author. In order to keep the different sessions coherent and comparable we followed a semistructured manuscript containing four themes we had selected as emerging concepts from our observations. We used pair-wise post-it discussions, followed by group discussions where each pair reflected on what they had come up with. The post-it stickers were collected, numbered and digitized. Each session was also recorded using video and sound.

2.8 Field experiments

Inspired the reasoning on *ethnographically natural* experiments by Hollan et al. [18], we decided to extend our data set with empirics from three minor field experiments. These were dressed as improvised exercises, which is a part of the overall course concepts, where *unplanned* customer changes could take place.

Field experiment – group constellation

Our first experiment consisted of creating four teams with different member compositions, with the purpose to see what differences, if any, we could observe during the observation study (and through the other data sources). See subsection 2.5.

Field experiment – exploratory testing

The second experiment consisted of assigning the students with a surprise story in preparation for the fifth sprint. The story consisted of little more than the instructions to: *execute roughly 1 hour of exploratory user tests of the system under realistic race conditions using four team members documenting the issues encountered*, and further to reflect on the experience in their weekly reflections (that all students fill out after each sprint). The story was handed out during the planning session the week before the full day sprint during which it was planned. We collected information of the activity from questionnaires (Q5/Q6) and from discussions with students and coaches during the following sprint and planning session.

Field experiment – merge-back

The third experiment consisted of the request to implement two sets of changes, in two separate files, and upon completion of the first task request a merge-back and recreation of the first release. Each team was handed a story card describing the two code blocks to be implemented *first thing in the morning* during the final planning session leading up to the final sprint. Each team was asked to notify their *customer* upon completion of the task. In order not to compromise that functionality/integrity of their respective systems the two code blocks were dummy snippets that were commented out. The experiment was documented using video and sound recording.

2.9 Analysis

Given that we had a limited time window for our observation, we did not have a lot of time for analysis during the field work. We exchanged notes and discussed our observations over lunch breaks. After the field work was completed the first and second author started a more formal analysis stage.

Initial coding [6] – the first and second author each did open line-by-line coding of the student reflections and the post-it stickers. We then exchanged our reflections in short memo form. In parallel, the first author did an initial overview of the contents of the questionnaires.

Focused coding [6] – the first and the second author had a two day session in which the questionnaires, focus groups post-it stickers and student reflections were analysed from multiple perspectives and the parts that we found relevant was extracted and documented digitally. We also extracted relevant 'soundbites' from free text answers, and digitised them. The findings were condensed in a short memo.

Theoretical coding [6] – the theoretical coding was executed by the first author, using Glaser's *'6C'-coding family* [12] as a starting point. The work was done in memo form and visualized on an A1 sheet using postit stickers. After a few iterations of coding, sketching and memoing a theory was emerging. The first and third author had a one hour session in which the theory was discussed from various angles and a few of the constructs were redefined. After this the first author did a minor rewrite of the theoretical coding memo.

2.10 Theoretical saturation

Having iterated through *open coding* and *focused coding* of the data set, we saw the need of further *saturation* in order to provide some more insight from *the members* *point of view*. In order to do so, we went through the recordings of the focus groups in order to provide some additional insight. Finally we reached out to a handful of students whom previously agreed to do minor follow up interviews. We held three short (15–20 minutes) open interviews specifically aimed at understanding what the students perceived as tool interaction related issues. The interviews were conducted by the first author and were documented by additional field-notes. All quotes and findings were reread to the subjects at the end of these interviews.

2.11 Literature review

In its original form, research questions in GT studies should emerge from the research, not be defined apriori [37] and *extensive* literature should be avoided prior to the emerging of theory. That being said, Charmaz takes a more pragmatic stance on literature and research questions and emphasises the iterative nature GT, thus allowing for initial research questions that evolve through the research project as well as abductive reasoning on extant literature, recommending a *preliminary* literature review '*without letting it stifle your creativity or strangle your theory*' [6].

As a consequence we did an initial, rather limited, literature study of Distributed Cognition from a Software Engineering perspective. Following the coding cycles we did an additional, or final, literature review on the central phenomenon of the theory we generated, i.e. GIT, version control and merge operations from a user perspective. See Section 4 for findings.

3 Analysis

This section present the theory generated from the dataset. Based on the findings from open and focused coding of our data set, the emerging concept we focused on was *issues regarding version control, branching and merge operations*.

For the first attempt at formulating the theory, a theoretical conceptual explanation of what we observed, we based our theoretical coding on Glasers *6 C-coding familly* [12] [37], while observing Thornberg and Charmaz reflection that the researcher should avoid being *hypnotized* by Glaser's coding families [41] – analogous to Glasers argument that all codes should *earn* [12] their way into the theory. Thus, we used *the 6 C's* as a starting point, and allowed for modifications throughout the *theoretical coding* phase.

A rendering of our conceptual model based on our analysis of the *issues regarding version control, branching and merge operations* encountered is illustrated in Figure 1. The center bottom rectangle describes the core phenomenon, *version control, branch & merge issues*, while the other codes are represented by surrounding rectangles. Cause, correlation and effect are represented by arrows. Context is represented using dotted arrows. For each code a corresponding subsection is found below. Along with the analysis, the conceptual model is detailed in Figure 2.

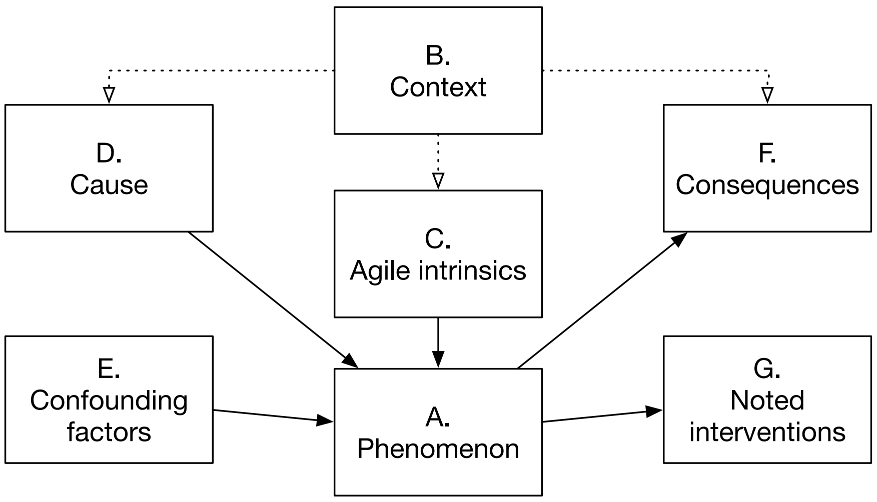

Figure 1: Conceptual model of the causal and consequential dimensions in regards to version control, branching and merge operations encountered in the projects.

Throughout the analysis section we provide examples of 'quotes' from the data set. 'S'/'I' denotes interview subject and researcher respectively. We have added *emphasis* for *clarity* and occasional further clarifications within [hard brackets].

3.1 Phenomenon

Throughout our observations (field notes) and our questionnaires we noted that version control, branching and merge operations caused a disproportionate amount of loss in productivity and time. The questionnaires for all teams systematically indicated *version control*, *branching* and *merge conflicts* as the most disruptive challenges encountered throughout the project, and as a consequence this is the phenomenon we chose to explore.

E.g.: 'Git/Merge – We are *unsure* of how to *use git properly*' (from student questionnaires – in response to what has been the biggest hurdle faced during the project).

3.2 Context

We define the context from which our observations are extracted, and in which they are valid as that of agile software development teams, consisting of novices,

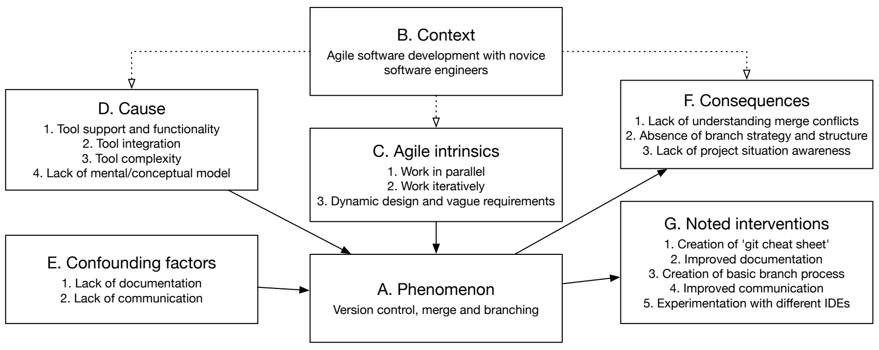
B. Context
Agile software development with novice software engineers
D. Cause
1. Tool support and functionality
2. Tool integration
3. Tool complexity
4. Lack of mental/conceptual model
C. Agile intrinsics
1. Work in parallel
2. Work iteratively
3. Dynamic design and vague requirements
F. Consequences
1. Lack of understanding merge conflicts
2. Absence of branch strategy and structure
3. Lack of project situation awareness
E. Confounding factors
1. Lack of documentation
2. Lack of communication
A. Phenomenon
Version control, merge and branching
G. Noted interventions
1. Creation of 'git cheat sheet'
2. Improved documentation
3. Creation of basic branch process
4. Improved communication
5. Experimentation with different IDEs

Figure 2: Conceptual model from Figure 1, further extended with the detailed codes from the analysis.

using GIT. Admittedly, this could result in a rather narrow validity window in terms of generalization. However, in our experience (both the first and second author has 15+ years experience of professional tool driven software development in large/distributed software projects) this observation, practitioners struggling with GIT, is commonplace in industry. Further, using novices as study objects would rather reveal cognitive challenges, as these challenges are not mitigated by *trained behavior*, *learning effect* or *status quo bias*.

We created our data set from observing and interacting with four different teams of *novice software developers* in parallel. All teams were using XP, and developed their system using the same basic stories/requirements (see Subsection 2.3 for details). While tool chain set up and development environment (IDE) differed somewhat between the teams, all teams used GIT hosted by Bitbucket for version control (albeit with different branch strategies).

In light of the observed lacunae in extant software engineering literature we note that version control from a user perspective is an area not thoroughly studied in the research community. Those few studies we found systematically indicate that our observations are valid in a wider context.

3.3 Agile intrinsics – root cause & driver

The iterative and parallel aspects of the nature of agile software development, *Agile intrinsics*, are from our observations, the identified underlying *Root Cause* of the observed merge conflicts. In order to achieve some granularity we further break this construct up into three different subcodes: *(Work) in parallel*, *(Work) iteratively* and *Dynamic design and vague requirements*, since they are related in terms of root cause but have quite different consequences. As indicated by the *intrin-*

sics in the main category, these traits are inherent (largely by design) in the nature of agile software development. While these root causes could be compressed into one code, we feel that they are not interchangeable and each deserve a closer description. For further clarity we added an additional subcode, *Observed driver*, as means to further clarifying the underlying nature of these codes.

(Work) in parallel – observed root cause

When starting up the project, the code base is very small, and different programming pairs are developing, and modifying the same code/classes/files s, creating dependencies and diverging implementations ultimately leading to merge conflicts. While this to some extent was mitigated by adopting rudimentary branch strategies, the problem persisted throughout the projects.

We note that it appeared hard for the developers to find out *who did what, when and why?*, ultimately leading to a lack of understanding of implementation details, or a micro perspective, thus making subsequent merge conflicts harder to resolve. We also noted that this caused the developers to implement their own variations of similar methods (e.g. utility methods). E.g.:

– ’Trying to merge code that someone else has written.’

(from student questionnaires and focus groups in response to what they found being difficult during their projects).

Observed driver: Diverging implementations leads to conflicting implementation details, further resulting in merge conflicts.

(Work) iteratively – observed root cause

The iterative nature of the development results in constantly shifting implementation details and this subsequently drives merge work. The constant change in code leads to a lack of understanding from a micro perspective, reimplementation and duplication of code as different development pairs reimplement existing functionality. E.g.:

– ’Parts of code *unknown*, having to interact with code that *someone else has written*, better after refactoring.’

(from student questionnaires and focus groups in response to what they found being difficult during their projects).

Observed driver: Refactoring implementation leads to changing implementation details, further resulting in merge conflicts.

Dynamic design and vague requirements – observed root cause

Since there is no set architectural design/framework, nor a complete set of requirement or user stories, in the beginning of the projects, there was no cohesive collective goal for the developers. Further, architectural changes drives extensive refactoring and results in subsequent merge conflicts. Despite the fact that this is an inherent feature of XP – 'XP is a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements' [2] – it is none the less something we noted as a systematic cause of refactoring and merge conflicts. E.g.:

– '*Hard* to change data structures. This causes merge conflicts and bugs. Improve communication [within the team]?'

(from student questionnaires and focus groups in response to what they found being difficult during their projects).

Observed driver: No set design at the beginning of projects leads to refactoring of structure and changing of architecture, resulting in merge conflicts

3.4 Observed cause

This part of the analysis provides a reasoning on our observations on the observed causes of merge incidents.

The impact of tool support and tool functionality

Throughout the study we noted that the students were quite opinionated about functionality and support of the different tools and how well they were integrated. All teams started their respective projects using Eclipse and GIT hosted on Bitbucket. Out of the four teams, two ultimately migrated from Eclipse to another IDE; intelliJ in one case and VSCode in one case.

When discussing tools during focus groups the importance of the user support the developers experienced became obvious. We noted that ease of use, intuitive interaction and visual support and offloading was something the students noted as very important in terms of reducing cognitive load. This is illustrated below in an excerpt from a focus group dialogue between three students (SI–SIII) and the interviewee (I):

SI: – 'Many had problems *seeing* what changes that were being made, that is when you fetch; it might be related to Eclipse [integration with git], it became better with VSCode, with the *colours* [indicating visual offloading], the *visual*, to be able to understand what has happened.'

I: – 'But what *experience* have you had in regards to the *tool support* you have had in order to *solve merge conflicts*?'

SI: – 'Eclipse was really *messy*.'

SII: – '...it was really hard to see *what* changes were coming from *where* – [extended pause, thinking] - and I think the *colours* in VSCode are really good [indicating visual support]. You really *see*, visually, *what* is *what*.'

SIII: – '...when we were using Eclipse we *switched to using the terminal* [use of GIT through command line interface (CLI) instead of IDE integration] instead, it just feels a lot *easier*.'

(Focus Group, excerpt from video recording T@12.43).

The importance of visual support became even more obvious during saturation:

S: – 'The merge support in VSCode is *graphical* and easy to understand – it is *intuitive*.'

S: – 'The *merge support* in VSCode is very *clear*, it provides help on resolving the conflict, it *shows* <source code 1> and <source code 2>in the GUI and it is *simple* to choose by clicking a button.'

S: – 'intelliJ is actually, in my opinion, better than VSCode. It gives even better and *more visual* merge support.'

(Field notes – saturation)

Tool integration

We decided to further break down the analysis of the tool support further, in order to be able differentiate different angles of the experience of the students. We noted that the actual integration of GIT in the IDEs was considered quite important, and a contributing factor when it came to changing IDE.

S: – 'The *graphical integration of GIT* in Eclipse is *difficult to understand.*'

S: – 'Eclipse is *complicated* in terms of GIT integration, and it is *easier to use* git through a *terminal* than through Eclipse.'

S: – 'The *integration* between git and VSCode is *superior* to that of Eclipse.'

(Field notes – saturation).

Tool complexity

The actual importance of tool complexity came to some surprise to the first author. We observed several reflections on the intricacies and complexities of GIT in the dataset. We found compelling evidence that the complexity of GIT was indeed a main cause of concern and cognitive load for novices, but the intricacies of GIT was not the only cause of concern – the complexity of the IDE was also a definite issue and cause of confusion.

S: – 'Version Control - GIT is very *difficult*.'

S: – 'What would make Eclipse better? Better *merge support* and better overview, making it empheasier to find functionality.'

S: – 'VSCode feels *simpler*, with less functionality but it is a lot *less overwhelming*. It has a lot better learning curve.'

S: – 'Eclipse is *complicated* and it is *difficult to understand* the structure.'

(Field notes – saturation).

Somewhat counter intuitively we also observed the following reflections on GIT the command line interface:

S: – 'GIT/CLI [in terminal] is good because it looks the same in every environment.'

S: – 'GIT/CLI [terminal] is good because all git online resources describe git through CLI, so it is a lot easier to copy a line of commands and paste it into the terminal than to try do do the same thing through a GUI.'

(Field notes – saturation).

Lack of mental/conceptual model of version control and branch structure

Based on the outcome of Experiment II, which we considered a trivial git/branch operation, we noted that the students' understanding of reasonably straight forward branch operations in GIT was somewhat limited. Out of the three groups that did the experiment (one team dropped out because of time constraints in their project), no one came up with a viable solution (albeit they came up with interesting and manually labour intensive ways to approach the task). At the end of the time-slot given for coming up with a solution, the first author provided a hint of the form 'Well, maybe you should google *git squash* and *git cherry pick*?'. Subsequently all three teams adequately solved the exercise in a matter of minutes.

3.5 Confounding factors

Lack of documentation

We noted that a systematic lack of documentation (i.e. *code comments*, *commit messages*, *design documentation*) plagued the groups throughout their respective projects. This added to the lack of understanding the merge conflicts. We also noted that the students became aware of these aspects and, to a varying degree of success, tried to adress these issues at the later stages of their projects, see Subsection 3.7. Because of space limitations and the secondary nature of this code we have omitted any actual quotes, but the issues were systematic and affected all teams.

Lack of communication

We noted that a systematic lack of communication within the team (e.g. *standup meetings* and use of *story boards*) plagued the groups throughout their respective projects. This added to the lack of understanding the merge conflicts as well as a lack of understanding the current project status. Further it added to *waste* and *loss of team productivity* when different pairs were working on the same task in parallel without knowing this. We also noted that the students became aware of these aspects and, to a varying degree of success, tried to address these issues at the later stages of their projects, see Subsection 3.7. Similar to the above, we omitted actual quotes, but the issues were systematic and affected all teams. One group started using Trello instead of a physical story wall, while the others continued using story walls.

3.6 Consequence

This part of the analysis provides a reasoning on our observations of consequences of the phenomenon under study.

Lack of understanding merge conflicts

The systematic lack of understanding of merge conflicts surprised us, and it became the focus of the analysis. These merge conflicts obviously lead to a loss of productivity, but it is not only limited to that. When going through the focus group material and the student reflections we saw multiple examples of negatively loaded wording, indicating *fear*, *insecurity* and *stress*. We find this to be clear indicators that issues with merge conflicts not only cause a loss of productivity in terms of linear time, but also that the absence of the needed tool support causes considerable cognitive load and stress on the developers.

S: – 'It is *frightening* with a Wall of Text – merge conflict/difference [indicating a very complicated merge] when in reality there is only a minor difference in a character or so [e.g. trailing space etc.]. In VS code you see both versions and you can simply choose what code [snippet] you want.'

S: – "You don't know how to revert changes in GIT you don't know if you will *accidentally* [loss of control] replace/delete something [important]... you need to *dare* to use GIT..."

S: – '*uncertainty* results in many [of us] finding it *stressful* with merge conflicts... when there is a "merge message" that just appears you don't really know what it means - will it result in overwrite - this makes it feel difficult, perhaps more so than it actually is...'

(from Field notes – saturation, questionnaires and focus group interaction).

Absence of branch strategy and structure

In addition to the systematic lack of understanding merge conflicts we also noted that branching itself was quite difficult for the teams. They had a hard time coming to grips with when to use separate branches (e.g. for bug fixes, tasks, stories and releases), when to close superfluous branches and branch naming conventions.

S: – 'It would have been better if we had used *story specific branches*.'

S: – 'We did not have a *strategy for branching* from the beginning [of the project].'

S: – 'We should have *closed branches* that were no longer in use.'

(from Field notes – saturation, questionnaires and focus group interaction).

Lack of project situation awareness

Further we noted that there were issues in regards to understanding the current project situation/status. This included multiple pair working on the same tasks, different pair implementing similar utility functions, a lack of understanding of components in the projects, and ultimately not knowing whom to ask about implementation details.

S: – 'Lack of communication - many of the problems we are facing would be solved if we would communicate better.'

S: – 'People working on the same issue – sometimes people work with solving the same problems without knowing it/each other.'

S: – 'Lack of communication – this lead to several interesting issues during sprint III where we went in different directions regarding architecture.'

(from Field notes – saturation, questionnaires and focus group interaction).

3.7 Noted interventions

We here describe the interventions implemented by the different teams as means to circumvent the issues they encountered in their projects. On account of space limitations we omit the qualitative excerpts and keep the description short.

Creation of 'git cheat sheet'

We noted that the teams, after the first few sprints, realized that they needed a common manual for (and understanding of) basic GIT operations. This was in most cases implemented as a *spike* by a pair of team members in between sprints. Further, we saw an interesting example of knowledge transfer within the team.

Improved documentation

We noted that the all teams throughout the project started realising the importance of documentation. The observed interventions included a systematic way of describing commits (i.e. pointing out what story or what task had been worked on, rather than the initial, rather void, messages like *'bugfix'*, *'gui implementation'* etc.). We also noted that the teams started documenting the design of their architectures (using UML) and user interfaces (sketching on A3 paper). In addition we also noted that, while struggling with it in practice, all teams realised the importance of code documentation and made considerable attempts at documenting their code properly.

Creation of basic process for branch/cm/releases

We noted that all teams, after a few sprints started to develop a basic branch and configuration management process. This consisted of a more rigorous – less ad hoc – naming convention of branches, systematisation of main branch integration, and use of separate branches for stories, amongst other things. We do not consider the actual details as important as the observation that the teams, themselves, organically came to the conclusion that they needed a more systematic approach in regards to branching and configuration management. In addition we also noted that all teams, having experienced the value of explorative testing in Experiment I, started doing so well in advance of their releases.

Improved communication

We noted that all groups became aware of the need of improved communication. One team started using Trello as means of establishing a sound project overview. All teams further noticed the importance of standup meetings, and systematically started running more frequently.

Experimentation with different IDEs

As previously described we noted that two of the teams started exploring other IDEs in order to circumvent their perceived issues with Eclipse.

4 Literature review

We will here briefly discuss our findings from extant literature in regards to Distributed Cognition, Git and Tool complexity.

4.1 Distributed cognition

Distributed cognition (DC) is a sub-discipline of studies of cognition in which the one of the traditional cornerstones of cognition – 'that cognitive processes such as memory, decision making and reasoning, are limited to the internal mental states of an individual' [15] – is questioned and rejected. Instead it argues that the social context of individuals as well as artifacts forms a cognitive system transcending the cognition of each individual involved [11], i.e., a cognitive system extending beyond the mind of one single individual [23]. The concept was pioneered by Hutchins who studied the cognitive activities on the navigation bridge of US naval vessels [21].

Hollan, Hutchins and Kirsh extended DC into the realm of Human-Computer Interaction (HCI) as well as to some extent into Software Engineering, stating that a distributed cognitive process (or system) is 'delimited by the functional relations among the elements that are part of it, rather by the spatial colocation of the elements', and that as a consequence 'at least three interesting kinds of distribution of cognitive processes become apparent: [a)] cognitive processes may be distributed across members of a social group[;] [b)] cognitive processes may involve coordination between internal and external (material or environmental) structure [and, c)] processes may be distributed through time in such a way that the products of earlier events can transform the nature of later events.' (reformatted but verbatim). [18]

Despite the fact that the theory of distributed cognition was suggested as a fruitful approach for investigating and explicating phenomenon related to software engineering several decades ago – Flor and Hutchins empirically studied pair-programming from a distributed cognition perspective as early as 1991 [11] – few examples exist of actual software engineering studies using distributed cognition as scientific lens. In 2014, Mangalaraj et al. [23] highlighted Sharp and Robinson [33], Hansen and Lyytinen [15] and Ramasubbu et al. [27] as 'the few notable exceptions' of extant software engineering research utilising Distributed Cognition. To this list we would like to add Walenstein [43], a recent study by Buchan et al [5] and other works by Sharp et al. [32] [35] [36] [31] [34].

4.2 Git & Merge

Our literature findings in regards to user experience of GIT were surprisingly limited. What we could find was three relevant papers: Church et al. [9], Perez & de Rosso [26] and de Rosso & Jackson [10].

We note that these papers, to some extent, validate our findings that GIT is a very complex tool to use, and our conclusion is that there is considerable lacunae in literature in this regard. Future research should include a more thorough literature study in regards to GIT and merge tools.

4.3 Eclipse and tool complexity

The issues related to tool complexity among novices are largely substantiated by extant literature – Moody [24] discussed the different levels of support needed by novices and experts when it comes to visual languages based on Cognitive Fit Theory [42]. We can also see the same patterns in research on expertise by Chi et al. [8]. Further, Storey et al. [38] have specifically described issues of novices in regards to Eclipse.

5 Ethical considerations

With the study focus on groups rather than individual students, there was no legal need for formal ethical hearing. We did submit and register a description of the study to ethics board. The course is graded *Pass* and *Fail* only, and the only way students to fail is by considerable abscence, we felt that there was no issue with conflicting roles of researcher/teacher for the first author. In addition, our presence during sprints and planning sessions allowed the student groups more teacher time than what they would have experienced otherwise.

The experiments we exposed the students to had been used as improvised *project disturbances* and as improvised exercises aimed at exploratory testing by the first author in previous years, and it appeared to make a sound addition to the learning outcome of the students. Based on the fact that the learning outcome of the students was not compromised, that all data was collected with consent and anonymously, and that the findings will benefit the students of the next instantiation of the course, we do not feel that we have any ethical qualms in regards to the study.

6 Threats to validity

The use of students as basis for research can be controversial [19] [40] [17] from a generalisation perspective as well as from student integrity and learning perspectives. In terms of generalisation, Höst et al. highlight that students working under life-like circumstances serve can function as a reasonable proxy for real life settings/practitioners [20]. In this study we selected students to capture a *novice point of view*, thus providing us with a different perspective of causes of cognitive load drivers.

We further argue that the observed similarities between the groups and that the rigor with which the case study was designed and executed further strengthens the findings.

GT studies are commonly evaluated based on the following criteria [6] [37]:

Credibility: *Is there enough data to merit claims of the study?* – This study relies on the data set from one case study. The data set includes interviews, focus groups, observations and wirtten reflections. The data set is quite extensive.

Originality: *Do the perspectives offer new insight?* – While cognitive load is not an unknown phenomenon in software engineering, we note that merge operations seem disproportianetly troublesome/difficult. We note a *reserach gap* when it comes research on version control and merge operations.

Usefulness: *Is the theory generated relevant for practitioners?* – This study generates a theory that offers one explanation of how merge operations and branch work becomes difficult in projects. This can be used for reasoning on cognitive load in software engineering. The main contribution, in our opinion, is the observation of merge phenomenon and version control issues and the corresponding *research gap*.

We take a pragmatic postpositivist [28] 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. We acknowledge that all qualitative knowledge is inherently constructed. That being said, the phenomena we study do arguably exist, albeit in an artificial context largely unbound by natural laws. If the phenomena did not exist, there would be little point in studying them, nor their consequences on the human mind.

7 Sensitizing concepts

The findings in relation to the first research goal, *to identify the most common cognitive load driver from the novice point of view*, was somewhat surprising. While we build our work on our previous identification of the temporal perspective [16], the *who, did what, when & why*, we were quite surprised to see how large the impact of version control and merge operations were on the students. We also find it interesting to see the importance of tool support and functionality, tool integration and tool complexity in agile software development. To us the most interesting observation is the importance of visual merge support. We also noted that absence of communication and documentation was a contributing and confounding factor. We also note the absence of research on version control as an indicator for further research.

In addition to the codes described in our theory, we also noted other indications of cognitive load drivers in the material. The environment, in terms of ventilation on loud ambience were lamented on, describing the work situation as *draining*. Further we also noted disruptions and task switching as a cause of concern – described as a *disruption of flow*.

We noted that distributed cognition, from our perspective, is indeed a sound lens for observing and analysing software development in agile teams and it is further interesting to note the reflections of *history enriched objects* and *temporal* cognitive dimensions made by Hollan et al. [18].

8 Future research

Further research in relation to this specific study will include a more focused study using direct observation of merge conflict resolution, and a benchmarking of a few popular IDEs in terms of version control integration and merge support, focusing on usability aspects. For a more long term perspective, *mental models* and the temporal dimension of cognitive load in relation to software development tools could be used as a stepping stone for further research in terms of cognitive support for software development. The observed lacunae in extant literature in regards to version control, to us, is an indication that there is indeed relevant research to be done in this area, and the need of a systematic mapping study. It would further be interesting to do a critical analysis of research on literature on agile software development, possibly using meta-ethnography in conjunction with grounded theory.

Acknowledgement

The authors wish to thank all the participating students for their invaluable contributions, course responsible for providing the opportunity to conduct the study. We further thank Softhouse[2] for providing time for the second author. The work described in this paper was conducted in the ELLIIT[3] strategic research environment.

[2]https://www.softhouse.se
[3]https://liu.se/elliit

References

[1] Steve Adolph, Philippe Kruchten, and Wendy Hall. Reconciling perspectives: A grounded theory of how people manage the process of software development. *Journal of Systems and Software*, 85(6):1269–1286, June 2012.

[2] Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

[3] Alan F. Blackwell, Marian Petre, and Luke Church. Fifty years of the psychology of programming. *Int. J. Hum. Comput. Stud.*, 131:52–63, 2019.

[4] Susanne Bødker. Third-wave hci, 10 years later - participation and sharing. *Interactions*, 22(5):24–31, 2015.

[5] Jim Buchan, Didar Zowghi, and Muneera Bano. Applying Distributed Cognition Theory to Agile Requirements Engineering. In Nazim Madhavji, Liliana Pasquale, Alessio Ferrari, and Stefania Gnesi, editors, *Requirements Engineering: Foundation for Software Quality*, Lecture Notes in Computer Science, pages 186–202, Cham, 2020. Springer International Publishing.

[6] Kathy Charmaz. *Constructing Grounded Theory*. SAGE Publications, London, UK, 2nd edition, 2014.

[7] Kathy Charmaz and Ricard Mitchell. Grounded Theory in Ethnography. In *Handbook of Ethnography*. SAGE Publications, London, UK, 2001.

[8] Michelene T. H. Chi, Robert Glaser, and Marshall J. Farr. *The Nature of Expertise*. Psychology Press, January 2014.

[9] Luke Church, Emma Soderberg, and Elayabharath Elango. A case of computational thinking: The subtle effect of hidden dependencies on the user experience of version control. In Benedict du Boulay and Judith Good, editors, *Proceedings of Psychology of Programming Interest Group Annual Conference*, pages 123–128, Brighton, United Kingdom, June 2014.

[10] Santiago Perez De Rosso and Daniel Jackson. Purposes, Concepts, Misfits, and a Redesign of Git. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 292–310, New York, NY, USA, 2016. ACM.

[11] Nick V. Flor and Edwin L. Hutchins. Analyzing distributed cognition in software teams: a case study of team programming during perfective maintenance. In Jürgen Koenemann-Belliveau, Thomas Glenn Moher, and Scott P. Robertson, editors, *Proceedings of Empirical Studies of Programmers*, pages 36–64, Norwood, NJ, USA, 1991. Ablex Publishing Corporation.

[12] Barney G. Glaser. *Theoretical Sensitivity*. Sociology Press, CA, USA, 1978.

[13] Barney G. Glaser. *Emergence vs Forcing - Basics of Grounded Theory Analysis*. Sociology Press, CA, USA, 1992.

[14] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory*. AldineTransaction, New Jersey, USA, 1967.

[15] Sean Hansen and Kalle Lyytinen. Distributed Cognition in the Management of Design Requirementsdistributed cognition in the management of design requirements. In Robert C. Nickerson and Ramesh Sharda, editors, *Proceedings of the 15th Americas Conference on Information Systems*, page 266, San Francisco, California, USA, August 2009. Association for Information Systems.

[16] Daniel Helgesson, Emelie Engström, Per Runeson, and Elizabeth Bjarnason. Cognitive Load Drivers in Large Scale Software Development. In *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '19, pages 91–94, Piscataway, NJ, USA, 2019. IEEE Press.

[17] Joseph Henrich, Steven J. Heine, and Ara Norenzayan. The weirdest people in the world? *Behavioral and Brain Sciences*, 33(2-3):61–83, June 2010.

[18] James Hollan, Edwin Hutchins, and David Kirsh. Distributed Cognition: Toward a New Foundation for Human-computer Interaction Research. *ACM Trans. Comput.-Hum. Interact.*, 7(2):174–196, June 2000.

[19] Martin Höst, Björn Regnell, and Claes Wohlin. Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering*, 5(3):201–214, November 2000.

[20] Martin Höst, Claes Wohlin, and Thomas Thelin. Experimental context classification: incentives and experience of subjects. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 470–478, St. Louis, MO, USA, May 2005. Association for Computing Machinery.

[21] Edwin Hutchins. *Cognition in the Wild*. MIT Press, 1995.

[22] Per Lenberg, Robert Feldt, and Lars Göran Wallgren. Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, 107:15–37, September 2015.

[23] George Mangalaraj, Sridhar Nerur, RadhaKanta Mahapatra, and Kenneth H. Price. Distributed Cognition in Software Design: An Experimental Investigation of the Role of Design Patterns and Collaboration. *MIS Quarterly*, 38(1):249–A5, March 2014.

[24] Daniel Moody. The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, November 2009.

[25] Peter Naur and Brian Randell. Software engineering: Report on a conference sponsored by the nato science committee. Technical report, Scientific Affairs Division, NATO, January 1969.

[26] Santiago Perez De Rosso and Daniel Jackson. What’s Wrong with Git?: A Conceptual Design Analysis. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 37–52, New York, NY, USA, 2013. ACM.

[27] Narayan Ramasubbu, Chris F. Kemerer, and Jeff Hong. Structural Complexity and Programmer Team Strategy: An Experimental Test. *IEEE Transactions on Software Engineering*, 38(5):1054–1068, September 2012.

[28] Colin Robson. *Real World Research*. Malden: Blackwell, 2nd edition, 2002.

[29] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, March 2012.

[30] Helen Sharp, Yvonne Dittrich, and Cleidson R. B. de Souza. The Role of Ethnographic Studies in Empirical Software Engineering. *IEEE Transactions on Software Engineering*, 42(8):786–804, August 2016.

[31] Helen Sharp, Rosalba Giuffrida, and Grigori Melnik. Information Flow within a Dispersed Agile Team: A Distributed Cognition Perspective. In *Agile Processes in Software Engineering and Extreme Programming*, Lecture Notes in Business Information Processing, pages 62–76. Springer, Berlin, Heidelberg, May 2012.

[32] Helen Sharp and Hugh Robinson. An Ethnographic Study of XP Practice. *Empirical Software Engineering*, 9(4):353–375, December 2004.

[33] Helen Sharp and Hugh Robinson. A Distributed Cognition Account of Mature XP Teams. In *Extreme Programming and Agile Processes in Software Engineering*, Lecture Notes in Computer Science, pages 1–10. Springer, Berlin, Heidelberg, June 2006.

[34] Helen Sharp and Hugh Robinson. Collaboration and co-ordination in mature eXtreme programming teams. *International Journal of Human-Computer Studies*, 66(7):506–518, July 2008.

[35] Helen Sharp, Hugh Robinson, and Marian Petre. The role of physical artefacts in agile software development: Two complementary perspectives. *Interacting with Computers*, 21(1-2):108–116, January 2009.

[36] Helen Sharp, Hugh Robinson, Judith. Segal, and Dominic Furniss. The role of story cards and the wall in XP teams: a distributed cognition perspective. In *AGILE 2006 (AGILE'06)*, pages 11 pp.–75, July 2006.

[37] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 120–131, May 2016.

[38] Margaret-Anne Storey, Daniela Damian, Jeff Michaud, Del Myers, Marcellus Mindel, Daniel German, Mary Sanseverino, and Elizabeth Hargreaves. Improving the usability of Eclipse for novice programmers. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, eclipse '03, pages 35–39, Anaheim, California, October 2003. Association for Computing Machinery.

[39] Margaret-Anne Storey, Neil A. Ernst, Courtney Williams, and Eirini Kalliamvakou. The who, what, how of software engineering research: a socio-technical framework. *Empirical Software Engineering*, 25(5):4097–4129, September 2020.

[40] Mikael Svahnberg, Aybüke Aurum, and Claes Wohlin. Using Students As Subjects - an Empirical Evaluation. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 288–290, New York, NY, USA, 2008. ACM.

[41] Robert Thornberg and Kathy Charmaz. Grounded theory and theoretical coding. In *The SAGE Handbook of Qualitative Data Analysis*. SAGE Publications, London, UK, 2014.

[42] Iris Vessey. Cognitive Fit: A Theory-Based Analysis of the Graphs Versus Tables Literature*. *Decision Sciences*, 22(2):219–240, 1991.

[43] Andrew Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. PhD thesis, School of Computing Sciecne, Simon Fraser University, 2002.

CHAPTER III

GROUNDED THEORY PERSPECTIVES OF COGNITIVE LOAD IN SOFTWARE ENGINEERING

Citation:
– working manuscript

Abstract

Background: The socio-technical characteristic of software engineering is acknowledged by many, while the technical side is still dominating the 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 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 step stone.

Results: We present a model 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 conceptual model as a starting point for 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 [4], where the technical side of the phenomena seems to be more studied than the social side [37]. There has been a thread of research with a social focus [56], and software engineering is acknowledged as an interdisciplinary field [39] 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 [52]. However, the scope of the program comprehension research is mostly limited to the programmer and and the program, while software engineering has a much broader scope.

While software engineering always has been [45], and still is [17] [49], concerned with efficiency and productivity, 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 [17]. 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' [49], where both *hard* and *soft* productivity factors are considered. Cognitive aspects discussed are primarily related to interruptions in the work environment [49, Ch.9], although cognitive load and psychological distress are presented as types of *waste* by Sedano et al. [49, Ch.19] [50]. 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* in an industrial case study [27], 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. To advance this line of research, we here survey software engineering literature and use a grounded theory approach to synthesize existing knowledge into a model for cognitive load in software engineering. We use *cognitive load theory* (CLT) by Sweller et al. [57] [58] as a step stone into the analysis, and also survey its impact in software engineering research.

The remainder of this paper is structured as follows: Section 2 presents CLT, critique on CLT and use of cognitive load in software engineering research. Section 3 covers grounded theory, research goals, data collection, analysis, literature review and a reflection on validity/generalisation issues. 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 concepts and future research.

2 Background

It has since Millers seminal paper 'The magic number seven plus/minus two' [40] 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 [50]. In an initial explorative case study [27] 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*, i.e. *causes of cognitive load*, 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 the program comprehension [52] 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 [23]. Further, the exploration of cognition pointed us towards two cognitive directions to further investigate and describe cognitive load, Distributed Cognition [31] [29] and Cognitive Load Theory [57] [58].

Following our first explorative case study, we conducted a larger comparative [48] case study [26] studying cognitive load drivers *from the novice point of view* by means of grounded theory ethnography [7] [8], using distributed cognition [31] [29] as scientific lens. The observed phenomenon we chose to pursue and explain was version control and merge operations which, to our surprise, was the largest cause of problems for all of the four 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 [57] [58], when learning complex tasks. Paas et al. 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' [46].

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) [57] [30] [50]. CLT is based on the model of working memory provided by

Baddeley [3] [2], assuming that information processing and storing are two separate, yet interdependent processes [2]. A central tenet in CLT is the assumption, analogously to Miller [40], 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 [15].

While using CLT [57] [58] serves as a step stone for 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 [54] (n=65) describing the similarities and differences between CLT and Human Computer Interaction (HCI) was executed by Hollender et al. [30] in 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. further highlight relevant criticism of CLT [30], namely that the principal components of cognitive load *intrinsic*, *extraneous* and *germane* cognitive load to some extent belong to different ontologies (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 [14]. The critique has elaborated by Moreno [42]. Debue & Van der Leemput [15] 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 *aggregative*, in the sense that cognitive load drivers each increase the aggregated cognitive load situation experienced by the subject. This is the very reason why we propose our model of cognitive load in software engineering. We argue that our current direction is valid, since it is possible to identify cognitive load drivers regardless of their nature in terms of being *additive* or *aggregative*, 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 Cognitive Load in Software Engineering Research

Cognitive load has to some extent been studied in software engineering contexts. Goncalez et al. [23] list 33 studies in a systematic mapping study from 2019 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 & Muller [19]). Program comprehension is one dimension of software engieering where cognitive load has been studied [52].

In their paper on *software development waste*, Sedano et al. describe a constructivist grounded theory study at a case company over two years, resulting in a taxonomy of software waste [50]. 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.

3 Method

3.1 Grounded theory

In this study we conduct a qualitative synthesis of three grounded theory studies and extant literature. 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 & Dybå [12] on research synthesis, and the same authors have since published guidelines, or recommendations, for synthetical activities [11]. The authors list 13 different methods [12] 'for the synthesis of qualitative and mixed method evidence' including meta-ethnography [18] and grounded theory [7] [55].

The authors describe grounded theory as: '...a primary research approach that describes methods for qualitative sampling, data collection, and data analysis. It includes simultaneous phases of data collection and analysis, the use of the constant comparison method, the use of theoretical sampling, and the generation of new theory. It treats study reports as a form of data on which analysis can be conducted to generate higher-order themes and interpretations'.

Grounded theory (GT) [22] originated in the mid sixties as a qualitative reaction to the positivistic and quantitative research paradigms at the time dominant in the social sciences [7] [61]. It has since gained considerable traction in many fields outside of social science, and diverged into three main currents [7] [55] – i.e. Glaserian (original/agnostic), Strauss & Corbin (interpretist) and Charmaz (constructivist).

Inherently inductive [7] (or abductive [38]) and iterative [7], the main purpose of GT is to allow for generation of theory[1] based on qualitative data, and it fosters

[1]From a more positivist perspective *theory* in grounded theory context equates *model*. Being active in Software Engineering we use the construct *model* to avoid misunderstanding. Since we are using grounded theory and wish to avoid *method slurring* [55] and conceptual confusion we want to be transparent on the matter.

the researcher to systematically iterate between data collection, analysis and theory generation [7].

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 [22] has been interpreted as no literature should be consulted prior to the analysis [7] [38]. 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*' [21] (p35). The position of Charmaz is that the researcher should start with a preliminary literature review [61] that should be used 'without letting it stifle your creativity or strangle your theory' [7, p308].

'The SAGE Handbook of Current Developments in Grounded Theory' contains several chapters on the use of literature and abductive reasoning in the grounded theory studies (e.g. [61] [38] [6] [60] [24]) indicating that while there is a will within the grounded theory research community to further extend the role of literature in grounded theory research, there is as of yet no consensus on how. Reflecting on the original GT dictum, or maxim [38], 'all is data' [55] – that 'the word *data* seems to mean whatever Glaser or Strauss arbitrarily choose it to mean' [1] we do not really see an issue with fusing empirical data with literature.

Martin provides a sound analysis of abductive reasoning in GT arguing that it is inherent in grounded theory, and further states that: 'Theoretical sensitivity in grounded theory is stalled in part by unresolved debates about the role of extant literature in grounded theory. Further, the tension makes the abductive reasoning processes in grounded theory less explicit.' [38]. Stol et al. [55] present guidelines on GT in software engineering, as well as a critical review of the use of GT in SE research context. While we attempt, largely, to adhere to their guidelines – we are certain will take some form of objection.

A recurring quote in GT on the matter, and use, of preexisting knowledge is that of Dey: '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.' [16]

We take a pragmatic postpositivist [47] 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 – the epistemological position should reflect the nature of phenomenon under study. We acknowledge that all qualitative knowledge is inherently constructed. That being said, the phenomena we study do arguably exist, albeit in an artificial context largely unbound by natural laws. If the phenomena did not exist, there would be little point in studying them, nor their consequences on the human mind.

3.2 Research goals

Central to Glasers *original* GT as well as to Charmaz *constructivist* GT is that the final research questions are not defined up front at the beginning of the research project. In the case of Glaser the researcher should start with *area of interest* [21] [55], in the case Charmaz the researcher should start with *initial research questions* that evolve through the study [7] [55].

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 present a grounded theory for reasoning on cognitive load in a software engineering context; and

B) To survey the impact of cognitive load theory in the software engineering research community.

3.3 Data collection

Study I (SI) – In the first case study [27], 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 [5].

The design of this study was not formulated as GT, but as an *explorative case study* [48]. 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 *saturation*.

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* [7].

Study II (SII) – In the second case study [26], we set out to chart *cognitive load drivers* from a *novice point of view* using grounded theory ethnography [8] [7]. 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 for saturation following the fist round of open coding.

This study was specifically designed as grounded theory ethnography, and largely adhered to the guidelines provided by Charmaz [7], in terms of data collection and analysis. The study used Distributed Cognition as scientific lens, compared four different teams and we went through stages of *open*, *focused* and *theoretical coding* [7], using *memos/memoing* [20] 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 theory generation.

3.4 Analysis

This paper originated as a memo describing *sensitizing concepts* [7] in regards to cognitive load in software engineering and CLT. Open, focused and theoretical [7] coding of the result of the previous two studies (SI, SII) and the findings of Sedano et al. [50] was done by the first author alternating between post-it stickers on whiteboards and memos [7].

Following the thoughts on abductive reasoning, presented by Martin [38] 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.

3.5 Literature review

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. used to chart CLT in HCI [30]. Hollender et al. limited their search to querying (spelling) ACM only and let result serve as a proxy, rather than completing an exhaustive search. We queried ACM Fulltext library and IEEE 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. [50] 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 timeframe. From a grounded theory perspective we find the concept of using two proxies (i.e. ACM & IEEE and a limited timeframe) the equivalent of *theoretical sampling* [38].

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 made a first rough exclusion, and the unclear papers were reexamined by the first and second author collectively.

3.6 Validity & Generalisation issues

Publishing *odd* qualitative findings in the software engineering research community can be a challenge [51]. Further Siegmund et al. note that there is far from consensus within the field on how to weigh internal and external validity [53].

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 [50] [7] [55]:

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 and wirtten reflections.

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. We 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* [50] reduction of cognitive load in software engineering can be seen as means to increase efficiency and productivity. If viewed as *cognitive work environment issues* [25] their reduction would equal improving the cognitive sustainability in software engineering.

In regards to external validity – grounded theory is a largely qualitative methodology, the findings are not statistical, and can not be statitically generalised. That being said, in study I we compared our findings to a general taxonomy of cognitive ergonomics [25]; in this study we compare our findings to those of Sedano et al. [50]. We do not see our findings as very particular to the context in which we have observed them.

In regards to internal validity – researcher bias [50] or prior knowledge bias [50] is likely are known to influence observations. We attempt to use bracketing, rigor and bias awareness in order to reduce these influences.

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

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 [50], 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 & Dibbern present a multiple case study investigating the importance of knowledge in software maintenance outsourcing [36].

4.1 Measuring cognitive load

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

Goncalez et al. conducted a systematic mapping study specifically addressing measuring cognitive load of software developers (2019) [23]. The authors found 33 papers, and provide a classification of the articles found. We note a certain overlap with our findings in this search, e.g. Muller & Fritz [44] and Crk et al. [10].

Fritz & Muller present two papers; on the use of sensor driven 'biometrics' to boost software developer productivity [19] and a case study aimed at predicting code quality online using various sensors [44].

Karras et al. used eye-tracking to study the impact of different linking variants of use cases and associated requirements on reading behaviour [32].

Crk et al present an empirical study in which programming expertise is explored using brain wave changes (EEG) [10].

4.2 Improving software development

We found two papers related to improvement of software engineering activities. Henley & Flemming present at tool for improving code change support in visual dataflow programming environments [28] while Moseler et al. present a prototype tool for visualising debugging scenarios [43].

4.3 Bordering on software engineering

Bordering on software engineering, Kelleher & Hnin describe an approach to predict the cognitive load of code puzzles [33]. In addition we found a proposed model for API learning by Kelleher & Ichinco [34], and an explorative analysis of the notational characteristics of decision models by Dangharska et al. [13].

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

CLT component	Perspective	SI	SII	SIII
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	

4.4 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. 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. (SIII) in order to establish a model 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. See Table 1 for an overview of the *perspectives* and how they are derived from field studies and their relation to CLT components. From the analysis, five implications for design of software engineering tools and practice, emerge as sensitizing concepts, which we present in footnotes, marked [SCn] and leave for future work.

5.1 Reflection on cognitive load and cognitive load theory

Analogously to *electrical load*, *cognitive load* is momentary, and conceptually *load* analogously translates into *power*. Over time this integrates into *power* or *energy*. As a consequence an increase in (unnecessary) cognitive load corresponds to a loss in cognitive productivity. Long term exposure to cognitive load leads to cognitive drain, something that is well known to equate (or imply) *unhealthy*. We wish however not only explore *cognitive productivity* and *cognitive amplification*, we also want to bring up *cognitive sustainability*.

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 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 [15] [30]. In critique on CLT by de Jong [14] and Moreno [42], 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.[2]

5.2 Intrinsic cognitive load

Sedano et al. aptly highlight that many (if not most) software development activities are *cognitively intensive* [50], 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* (SIII). We suggest this 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 observed that the cognitive task it self can be a cognitive load driver, for instance in 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.[3]

We thus continue our reflection on CLT by suggesting a:

[2]SC1 – We observe that the concept of *cognitive productivity* represents the software development organization's strive for productivity, while *cognitive sustainability* addresses the developers' wellbeing, which indirectly, of course, also affects the development organization.

[3]SC2 – 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.

Task perspective

A task centric perspective of cognitive load in software engineering is warranted by the *cognitively intensive* [50] nature of software engineering. We conlude that *task/-s too complex* as observed by Sedano et al. [50] 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 and automation of schemata for the task at hand [15] – 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 [9]. Similarly novices and expert have different needs in terms of cognitive support in digital tools [41] [62], 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 have 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. (SIII) noted noted *psychological distress* [50] 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 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' [59]. Kirsh has described the consequences of Cognitive Overload in a general workplace setting [35].[4]

We do not really see that these dimensions of cognitive load are not 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.

Environmental perspective

An environmental centric perspective of cognitive load in software engnieering 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* [50]. Given that the human bandwidth for cognitive load is limited [40], 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. 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 as a consequence they see *extraneous cognitive load* as *waste* [50]. They further use *Extraneous Cognitive Load* as a catch all element in

[4]SC3 – 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.

their waste taxonomy, containing *technical debt*, *inefficient tooling*, *waiting/multi-tasking*, *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*.

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. (SIII) 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*.

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. [50] present *Ineffective Communication* as one specific type of *software development waste*, but we choose to distinguish between *information* and *communication* in this model – 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*).[5]

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.. As a consequence, we find the tool centric perspective quite important.

Sedano et al. (SIII) 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* and thus serves as an example on how different the driver 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 forced 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

[5]SC4 – 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*.

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.

Communication perspective

A communication centric perspective of cognitive load in software engineering is derived from equally from (SII) and (SIII). The phenomena under study are associated to the *process of distributing information* rather than to the *nature of information itself.* Sedano et al. 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' [50].

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 realted 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.

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 [49, Ch.9]. Sedano et al. noted *unnecessary context switching* [50] as one aspect of *extraneous cognitive load*', while presenting *waiting/multitasking* as another type of *waste* in software engineering, outside of *extraneous cognitive load* (SIII). 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 reports on interruptions in software engineering in a 2010 case study [59], 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. the higher the intrinsic cognitive task load will correspond to 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'.

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 still eludes us. 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 [29], 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 tranformation 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 (*metainformation* and *essential information*) was produced at an earlier stage, possibly by someone else.[6]

[6]SC5 – We noted that merge operations, a temporal synthesis of metainformation (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 metainformation appears to have a higher intrinsic cognitive task load than production of either metadata or essential information, provided that the level of abstraction is comparable. To us this is an indication that, while the additional cognitive user

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?*

6 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 (SIII) [50]. 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 theory does not have any major impact.

While working on memos for this manuscript we noted some *sensitizing concepts* [7] from a design science perspective. Space limitations does not allow for an in depth reasoning, so they are described in footnotes 2–6, left for further research, which also extends into:

- an in depth study of literature on cognitive perspetive in regards to version control and merge tools.
- a study focused on benchmarking existing git integrations in a few a the existing IDEs
- a 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

Acknowledgement

The authors whish 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. The work described in this paper was conducted in the ELLIIT[7] strategic research environment.

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.

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

References

[1] Mats Alvesson and Kaj Sköldberg. *Reflexive Methodology - New Vistas for Qualitative Research*. SAGE Publications, London, UK, 3rd edition, 2018.

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

[3] Alan D. Baddeley. *The Psychology of Memory*. Basic Books, New York, NY, USA, 1976.

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

[5] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, January 2006.

[6] Antony Bryant and Kathy Charmaz. Abduction: The Logic of Discovery of Grounded Theory - An Updated Review. In *The SAGE Handbook of Current Developments in Grounded Theory*. SAGE Publications, London, UK, 2019.

[7] Kathy Charmaz. *Constructing Grounded Theory*. SAGE Publications, London, UK, 2nd edition, 2014.

[8] Kathy Charmaz and Ricard Mitchell. Grounded Theory in Ethnography. In *Handbook of Ethnography*. SAGE Publications, London, UK, 2001.

[9] Michelene T. H. Chi, Robert Glaser, Marshall J. Farr, Robert Glaser, and Marshall J. Farr. *The Nature of Expertise*. Psychology Press, January 2014.

[10] Igor Crk, Timothy Kluthe, and Andreas Stefik. Understanding Programming Expertise: An Empirical Study of Phasic Brain Wave Changes. *ACM Transactions on Computer-Human Interaction*, 23(1):1–29, February 2016.

[11] Daniela S. Cruzes and Tore Dybå. Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 275–284, September 2011. ISSN: 1949-3789.

[12] Daniela S. Cruzes and Tore Dybå. Research synthesis in software engineering: A tertiary study. *Information and Software Technology*, 53(5):440–455, May 2011.

[13] Zhivka Dangarska, Kathrin Figl, and Jan Mendling. 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)*, pages 1–9, September 2016. ISSN: 2325-6605.

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

[15] Nicolas Debue and Cécile van de Leemput. What does germane load mean? An empirical contribution to the cognitive load theory. *Frontiers in Psychology*, 5, 2014.

[16] Ian Dey. *Qualitative Data Analysis: A user-friendly guide for social scientists*. Routledge, London, UK, 1993.

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

[18] Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9):833–859, August 2008.

[19] Thomas Fritz and Sebastian C. Müller. Leveraging Biometric Data to Boost Software Developer Productivity. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 66–77, March 2016.

[20] Barney G. Glaser. *Theoretical Sensitivity*. Sociology Press, CA, USA, 1978.

[21] Barney G. Glaser. *Emergence vs Forcing - Basics of Grounded Theory Analysis*. Sociology Press, CA, USA, 1992.

[22] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory*. AldineTransaction, New Jersey, USA, 1967.

[23] Lucian Gonçales, Kleinner Farias, Bruno da Silva, and Jonathan Fessler. Measuring the Cognitive Load of Software Developers: A Systematic Mapping Study. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 42–52, May 2019. ISSN: 2643-7171.

[24] Andrea Gorra. Keep your Data Moving: Operationalization of Abduction with Technology. In *The SAGE Handbook of Current Developments in Grounded Theory*. SAGE Publications, London, UK, 2019.

[25] Jan Gulliksen, Ann Lantz, Åke Walldius, Bengt Sandblad, and Carl Åborg. Digital arbetsmiljö, en kartläggning (RAP 2015:17). Technical report, 2015.

[26] Daniel Helgesson, Daniel Appelquist, and Per Runeson. A grounded theory of cognitive load drivers in agile software development. In *manuscript in progress*, 2021.

[27] Daniel Helgesson, Emelie Engström, Per Runeson, and Elizabeth Bjarnason. Cognitive Load Drivers in Large Scale Software Development. In *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '19, pages 91–94, Piscataway, NJ, USA, 2019. IEEE Press.

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

[29] James Hollan, Edwin Hutchins, and David Kirsh. Distributed Cognition: Toward a New Foundation for Human-computer Interaction Research. *ACM Trans. Comput.-Hum. Interact.*, 7(2):174–196, June 2000.

[30] Nina Hollender, Cristian Hofmann, Michael Deneke, and Bernhard Schmitz. Integrating cognitive load theory and concepts of human–computer interaction. *Computers in Human Behavior*, 26(6):1278–1288, November 2010.

[31] Edwin Hutchins. *Cognition in the Wild.* MIT Press, 1995.

[32] Oliver Karras, Alexandra Risch, and Kurt Schneider. 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*, pages 2–12, Christchurch, New Zealand, 2018. ACM Press.

[33] Caitlin Kelleher and Wint Hnin. Predicting Cognitive Load in Future Code Puzzles. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*, pages 1–12, Glasgow, Scotland Uk, 2019. ACM Press.

[34] Caitlin Kelleher and Michelle Ichinco. Towards a Model of API Learning. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 163–168, October 2019. ISSN: 1943-6106.

[35] David Kirsh. A Few Thoughts on Cognitive Overload. *Intellectia*, (30):19–51, 2000.

[36] Oliver Krancher and Jens Dibbern. Knowledge in Software-Maintenance Outsourcing Projects: Beyond Integration of Business and Technical Knowledge. In *2015 48th Hawaii International Conference on System Sciences*, pages 4406–4415, January 2015. ISSN: 1530-1605.

[37] Per Lenberg, Robert Feldt, and Lars Göran Wallgren. Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, 107:15–37, September 2015.

[38] Vivian Martin. Using Popular and Academic Literature as Data for Formal Grounded Theory. In *The SAGE Handbook of Current Developments in Grounded Theory*. SAGE Publications, London, UK, 2019.

[39] Daniel Méndez Fernández and Jan-Hendrik Passoth. Empirical software engineering: From discipline to interdiscipline. *Journal of Systems and Software*, 148:170–179, February 2019.

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

[41] Daniel Moody. The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, November 2009.

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

[43] Oliver Moseler, Michael Wolz, and Stephan Diehl. Visual Breakpoint Debugging for Sum and Product Formulae. In *2020 Working Conference on Software Visualization (VISSOFT)*, pages 133–137, September 2020.

[44] Sebastian C. Müller and Thomas Fritz. Using (Bio)Metrics to Predict Code Quality Online. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 452–463, May 2016. ISSN: 1558-1225.

[45] Peter Naur and Brian Randell. Software engineering: Report on a conference sponsored by the nato science committee. Technical report, Scientific Affairs Division, NATO, January 1969.

[46] Fred Paas, Alexander Renkl, and John Sweller. Cognitive Load Theory: Instructional Implications of the Interaction between Information Structures and Cognitive Architecture. *Instructional Science*, 32(1-2):1–8, January 2004.

[47] Colin Robson. *Real World Research*. Malden: Blackwel, 2nd edition, 2002.

[48] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, 2012.

[49] Caitlin Sadowski and Thomas Zimmermann, editors. *Rethinking Productivity in Software Engineering*. Apress, Berkeley, CA, 2019.

[50] Todd Sedano, Paul Ralph, and Cecile Péraire. Software Development Waste. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 130–140, May 2017.

[51] Helen Sharp, Yvonne Dittrich, and Cleidson R. B. de Souza. The Role of Ethnographic Studies in Empirical Software Engineering. *IEEE Transactions on Software Engineering*, 42(8):786–804, August 2016.

[52] Janet Siegmund. Program Comprehension: Past, Present, and Future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 13–20, March 2016.

[53] Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on Internal and External Validity in Empirical Software Engineering. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 9–19, May 2015.

[54] Hannah Snyder. Literature review as a research methodology: An overview and guidelines. *Journal of Business Research*, 104:333–339, November 2019.

[55] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 120–131, May 2016.

[56] Margaret-Anne Storey, Neil A. Ernst, Courtney Williams, and Eirini Kalliamvakou. The who, what, how of software engineering research: a socio-technical framework. *Empirical Software Engineering*, 25(5):4097–4129, September 2020.

[57] John Sweller and Paul Chandler. Why Some Material Is Difficult to Learn. *Cognition and Instruction*, 12(3):185–233, September 1994.

[58] John Sweller, Jeroen J. G. van Merrienboer, and Fred G. W. C. Paas. Cognitive Architecture and Instructional Design. *Educational Psychology Review*, 10(3):251–296, September 1998.

[59] Edward R Sykes. Interruptions in the workplace: A case study to reduce their effects. *International Journal of Information Management*, page 10, 2011.

[60] Iddo Tavory and Stefan Timmermans. Abductive Analysis and Grounded Theory. In *The SAGE Handbook of Current Developments in Grounded Theory*. SAGE Publications, London, UK, 2019.

[61] Robert Thornberg and Ciaran Dunne. Literature Review in Grounded Theory. In *The SAGE Handbook of Current Developments in Grounded Theory*. SAGE Publications, London, UK, 2019.

[62] Iris Vessey. Cognitive Fit: A Theory-Based Analysis of the Graphs Versus Tables Literature*. *Decision Sciences*, 22(2):219–240, 1991.