A Grounded Theory of Cognitive Load Drivers in Novice Agile Software Development Teams

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

Daniel Appelquist
Softhouse AB
daniel.appelquist@softhouse.se

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

Abstract

Objective: The purpose of this paper is to identify the largest cognitive challenges faced by novices, developing software in teams, using distributed cognition as an observational filter.

Paradigm: Design science

Epistemology: Pragmatist

Methodology: Case study

Method: Using **grounded theory**, **ethnography** and **multi method data collection**, we conducted an observational study for two months following four 10-person novice agile teams, consisting of computer science students, tasked with developing software systems.

Result: This paper identifies version control and merge operations as the largest challenge faced by the novices, and provides a substantive theory generated from our empirical data explaining the observed phenomena. 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

By now it is well known that software development is a sociotechnichal phenomenon, rather than a purely technical-technological one (Bertelsen, 1997). It is further well known that most, if not all software development actvities are cognitively intensive (Sedano, Ralph, & Péraire, 2017) and rely on software development tools. The human cognitive ability to process and harbour information is limited (Miller, 1956), so for seemingly obvious reasons it would make sense to lessen the cognitive load on the individual user.

But inspite of more than 50 years of investigation in regards to psychological and cognitive dimensions and phenomena (Blackwell, Petre, & Church, 2019) softer, non-technical phenomena remain underinvestigated (Lenberg, Feldt, & Wallgren, 2015). Some twenty years ago Walenstein (2002) highlighted the need of cognitive design theory as means to inform software development tool design and suggested distributed cognition as a suitable foundation for such theory (Abend, 2008) building and theorization processes.

In an attempt to more broadly understand the cognitive load induced from software development tools and processes, we studied cognitive load drivers in large scale software development (Helgesson, Engström, Runeson, & Bjarnason, 2019) 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. We have further described a set of 'perspectives' (Helgesson & Runeson, 2021) from which cognitive load in software development can be observed and analysed.

While Walenstein (2002) described software development from a distributed cognitive perspective, distributed, agile, software development software development has changed considerably over the past

twenty years, so in order to further advance the understanding of cognitive load in software engineering/development and agile software development as a distributed cognitive set of phenomena, we set out to ethnographically (Sharp, Dittrich, & de Souza, 2016) study cognitive load drivers in agile software development projects, using grounded theory (Charmaz, 2014) in conjunction with distributed cognition (Hollan, Hutchins, & Kirsh, 2000).

As we hypothesise that some of the cognitive loads are compensated and mitigated through increased experience and workarounds learned over the years, we choose to study novice software engineers (Höst, Wohlin, & Thelin, 2005) in order to capture the novice *point of view* (Sharp et al., 2016). 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, 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 are the dominant load factors in the projects observed, and subsequently explore these phenomenons in detail. The remainder of the paper is arranged as follows: background, method, analysis, literature review review, ethical consideration, validity and discussion.

2. Background

2.1. Distributed Cognition

Not only is software engineering, as previously mentioned, a sociotechnical (Bertelsen, 1997) phenomenon, it is also 'cognitively intensive'(Sedano et al. 2017). If we allow ourselves to theorize (Abend, 2008) in regards agile software development in teams, it is quite easy to envision the phenomenon as a distributed network of cognitive agents solving cognitively loaded tasks using computers and software development tools[1].

Distributed cognition 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" (Hansen & Lyytinen, 2009) – is questioned and rejected. Instead it argues that the social context of individuals as well as artefacts forms a cognitive system transcending the cognition of each individual involved (Flor & Hutchins, 1991), i.e., a cognitive system extending beyond the mind of one single individual (Mangalaraj, Nerur, Mahapatra, & Price, 2014). The concept was pioneered by Hutchins who studied the cognitive activities on the navigation bridge of US naval vessels (Hutchins, 1995).

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 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). (Hollan et al., 2000)

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

[1]The dissertation (Walenstein, 2002) makes for excellent indepth reading on the matter

and Hutchins empirically studied pair-programming from a distributed cognition perspective as early as 1991 (Flor & Hutchins, 1991) – few examples exist of actual software engineering studies using distributed cognition as theoretical underpinning. Mangalaraj et al. (2014) highlighted Sharp and Robinson (2006), Hansen and Lyytinen (2009), and Ramasubbu, Kemerer, and Hong (2012) as "the few notable exceptions" of extant software engineering research utilising Distributed Cognition. To this list we would like to add Walenstein (2002), a recent study by Buchan, Zowghi, and Bano (2020) as well as Sharp and Robinson (2004), Sharp, Robinson, and Petre (2009), Sharp, Robinson, Segal, and Furniss (2006), Sharp, Giuffrida, and Melnik (2012), Sharp and Robinson (2008) and Zaina, Sharp, and Barroca (2021).

A recent "exploratory literature review" on "Cognition and Distributed Cognition" is presented by Begum (2021), where the authors reached a similar conclusion to ours – that despite its' intrinsic promises distributed cognition remains largely unused in software engineering.

3. Method

3.1. Grounded theory

Grounded theory (GT) is a systematic and rigorous methodological approach for inductively generating theory from data (Glaser & Strauss, 1967) (Charmaz, 2014) (Stol, Ralph, & Fitzgerald, 2016). Stemming from social sicences, 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 (Stol et al., 2016). It is iterative and explorative (Charmaz, 2014) in nature, and thus suitable for answering open ended questions such as *what's going on here?* (Stol et al., 2016).

We primarily opted for Charmaz GT handbook (Charmaz, 2014) as guidelines, using Bryant (2017) as a complementary perspective (in addition we also consulted earlier works by Glaser (1978),(1992)), specifically using grounded theory in conjunction with ethnography (Charmaz & Mitchell, 2001) – an approach that gives *"priority to the studied phenomenon or process – rather than the setting itself"* (Charmaz, 2014). The ethnographic approach allows for exploring not only *what* practitioners do, but also *why* they do it (Sharp et al., 2016). Core elements in the ethnographic approach is the empathic approach *to describe another culture from the members point of view* and the intrinsic *analytical stance* (Sharp et al., 2016). As with grounded theory, modern ethnography also stems from social sciences (Sharp et al., 2016). Not extensively used in Software Engineering (Sharp et al., 2016), it has however been used to study agile teams (Sharp & Robinson, 2006)(Sharp & Robinson, 2004).

3.2. Research goals

Central to 'original' Glaserian GT and Charmaz Constructivist GT is that the actual/final research questions are not defined up front. Glaser suggests that the researcher should start with an *area of interest* (Glaser, 1992) (Stol et al., 2016), while Charmaz suggestion is that the researcher should start with *initial research questions* that *evolve* through the study (Charmaz, 2014) (Stol et al., 2016). 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*.

3.3. Case description

The course that we used as study object is a mandatory course for sophomore computer science[2] students aiming at teaching practical software development in teams using agile methodology, presented in detail by Hedin, Bendix, and Magnusson (2005). The course runs for two terms (14 weeks) and consists of

[2]Translations of educations are difficult, in international terminology 'Computer Science' is as close as we can translate it. It is a five year master (engineering) program mostly aimed at software rather than hardware. The program resides at the Faculty of Engineering, and the program responsibles reside at the department of Computer Science.

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 (Sharp & Robinson, 2004) 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 *customers*, 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.

3.4. Design considerations

We opted for a flexible case study design (Runeson, Höst, Rainer, & Regnell, 2012), 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 (Sharp et al., 2016) 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* (Hollan et al., 2000) as initial lens, or filter, for our observations. Distributed cognition, further described in Section 5, is a branch of cognition studying cognitive processes distributed in groups rather than cognition from the individual perspective. While the use of an initial lens could be thought of by some readers as contradictory to the central tenet in GT, we hold this (potential) critique as moot. We were targeting observations of cognitive load drivers in interconnected network of people and digital tools, so we needed some starting point for our observations.

3.5. 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.

3.6. 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. For further ethical considerations, see Section 6.

3.7. 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. We specifically opted to not be part of breaks, lunch hour etc. for respect of the students integrity. Since our research focus is the phenomenon of cognitive load from a team perspective, rather than team work in general, we do not see this as a threat to our observations.

In addition, we added a weekly questionnaire to be filled out by each student after every sprint (all in all 4*10*6 = 240 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 perceive as the major problems they had been challenged by throughout the project. In the fifth questionnaires questions were added to capture the outcome of one of the experiments, see Subsection 3.8.2. 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.

3.8. Field experiments

Inspired the reasoning on *ethnographically natural* experiments by Hollan et al. (2000), we decided to extend our data set with the result and observations from three minor field experiments. These were dressed as improvised exercises, a part of the overall course concepts, where *unplanned* customer changes could take place (Hedin et al., 2005). One of them had been used by first author in previous years, the others were new.

3.8.1. 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 3.5.

3.8.2. 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.

3.8.3. 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.

3.9. 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 (Charmaz, 2014) – the first and second author each performed 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 (Charmaz, 2014) – 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 (Charmaz, 2014) – the theoretical coding was executed by the first author, using Glaser's *'6C'-coding family* (Glaser, 1978) 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.

3.10. 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.

3.11. Literature review

In its original form, research questions in GT studies should emerge from the research, not be defined apriori (Stol et al., 2016) 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*" (Charmaz, 2014).

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

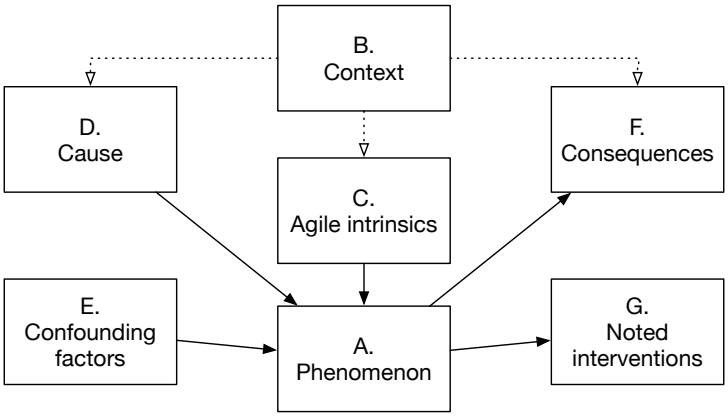

Figure 1 – Generated theory of the causal and consequential dimensions in regards to version control, branching and merge operations encountered in the projects.

review on the central phenomenon of the theory we generated, i.e. Git, version control and merge operations from a user perspective. See Section 5 for findings.

4. Analysis

This section presents 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 family* (Glaser, 1978) (Stol et al., 2016), while observing Thornberg and Charmaz reflection that the researcher should avoid being *hypnotized* by Glaser's coding families (Thornberg & Charmaz, 2014). This is analogous to Glasers argument that all codes should *earn* (Glaser, 1978) their way into the theory. Thus, we used *the 6 C's*[3] makes for excellent indepth reading on the matter as a starting point, and allowed for modifications throughout the *theoretical coding* phase.

A conceptual rendering of our generated theory 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 theory is detailed in Figure 2.

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].

4.1. 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.

[3]First author note – this was my first real attempt at 'pure' grounded theory. Having made another couple of attempts, to a varying degree of success, I make the casual observation that a 'grounded theory' consisting of more than 6-7 elements/constructs is very hard to explain to an audience. This is a perfect analogy to/example of Miller (1956). It is very hard for the human mind to process more than 6-7 visual elements simultaneously. So, if there are more constructs than the *the magical number seven* I humbly suggest breaking up the theory in subsets. If not, it will be very hard to convey the message visually – and it will in all likelihood be very hard to push through peer review...

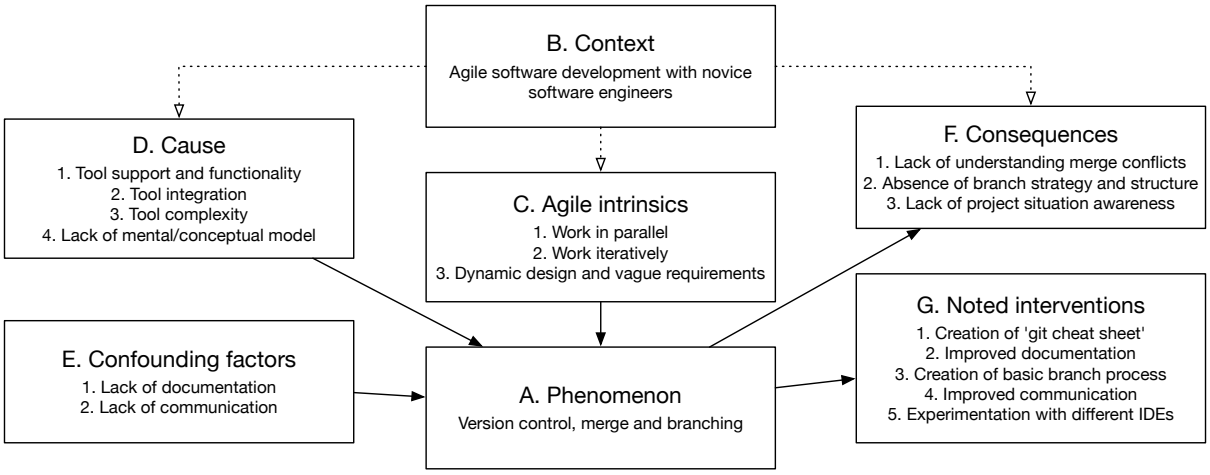

Figure 2 – Generated theory from Figure 1, further extended with the detailed codes from the analysis.

– 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).

We note that a *merge operation* in essence consists of a synthetical operation in which multiple sources/instances of software code is synthesised into a new instance, in an abductive process similar to what Walenstein (2002) describes as "the gulf of synthesis".

4.2. 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, 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 3.3 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.

4.3. 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 *intrinsics* 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.

4.3.1. (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, 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.

4.3.2. (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.

4.3.3. Dynamic design and vague requirements

Observed root cause: Since there is no set architectural design/framework, nor a complete set of requirements 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" (Beck, 1999) – it is nonetheless 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

4.4. Observed cause

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

4.4.1. 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)

4.4.2. 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).

4.4.3. 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 *easier 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).

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

Based on the outcome of the merge experiment (Subsection 3.8.3), 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.

4.5. Confounding factors

4.5.1. 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 4.7. 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.

4.5.2. Lack of communication

We noted that a systematic lack of communication within the team (e.g. abscence of *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 4.7. 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.

4.6. Consequence

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

4.6.1. 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).

4.6.2. 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).

4.6.3. 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: – 'Architecture – attempts to communicate architecture changes during iteration without documentation resulted in a loss of micro perspective Only after an architecture spike was the issue finally resolved and understanding was shared.'
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).

4.7. 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.

4.7.1. 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.

4.7.2. 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.

4.7.3. 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 the experiment presented in Subsection 3.8.2, started doing so well in advance of their releases.

4.7.4. 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.

4.7.5. 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.

5. Literature review

5.1. Git & Merge

Our literature findings in regards to user experience of Git were surprisingly limited. What we could find was three relevant papers: Church, Soderberg, and Elango (2014), Perez De Rosso and Jackson (2013), and De Rosso and Jackson (2016).

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.

5.2. Eclipse and tool complexity

The issues related to tool complexity among novices are largely substantiated by extant literature – Moody (2009) discussed the different levels of support needed by novices and experts when it comes to visual languages based on Cognitive Fit Theory (Vessey, 1991) (Vessey & Galletta, 1991) (Shaft & Vessey, 2006). We can also see the same patterns in research on expertise by Chi et al. (1981) (2014). Further, Storey et al. (2003) as well as Rigby and Thompson (2005) have specifically described issues of novices in regards to Eclipse.

6. Ethical considerations

With the study focus on groups rather than individual students, there was no legal need for formal ethical hearing under the jurisdiction under which this research was conducted, however we did submit and register a description of the study to the local ethics board. As the course is graded *Pass* and *Fail* only, and the only way students to fail is by considerable absence, we felt that there was no major 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. Further, we stress again that all students were systematically offered to retire themselves from the groups being observed.

Liebel and Chakraborty (2021), present an updated mapping study on ethical issues in empirical software engineering studies using students, and highlight that study conditions and power relations between students and instructors are special areas of concern. We would like to stress that the consent to enter the study was informed, and we feel that we presented all students with the systematic ability to retire from the research, that we have been transparent with the conditions and that the power relations were, in reality, unaffected by the study condition since the role of the researchers were to act as customers in the actual projects.

The experiments we exposed the students to had been used as improvised *project disturbances* embedded in the course design by first author in previous years and 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, that the findings will benefit the students of the next instantiation of the course and the very high course evaluation grades the students awarded the course post completion, we do not feel that we have any ethical qualms in regards to the study.

7. Threats to validity

The use of students as basis for research can be controversial (Höst, Regnell, & Wohlin, 2000) (Svahnberg, Aurum, & Wohlin, 2008) (Henrich, Heine, & Norenzayan, 2010) 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 (Höst et al., 2005). 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. Further, by acting as customers on site we were able to take a participatory observation position allowing us to some extent 'blend', while retaining an analytical ethnographic stance (Sharp et al., 2016).

In addition to discussing ethical dimensions of software engineering carried out on student populations, Liebel and Chakraborty (2021) also discuss the scientific value of such studies. Highlighting that research conducted using qualitative methodologies such as *case studies, observational studies and ethnography* the actual *case context* is a "deciding factor" and therefore cannot generally be separated from the "studied phenomenon". That being said, Stol and Fitzgerald (2018) highlight the value of knowledge seeking research approaches such as ethnography using the work by Sharp and Robinson (2004) as an example of such research.

Somewhat tounge-in-cheek (and not drilling into the taxonomy of elephants, which is extensive and contextually important), we respond to the elephant/jungle metaphors provided by Stol and Fitzgerald (2018) by stating that if you want to study juvenile elephants ridding themselves of bug infestation, and the methodologies deployed in such an activity you probably want to do it in the jungle[4]. If you, on the other hand, want to observe software development teams consisting of juniors solving software development issues, a computer hall at a university is, probably, about as ideal (and actually natural) as research environments come.

The procedures for the planning, data collection and analysis are reported in detail in Section 3.

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

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 written reflections. The data set is quite extensive.

Originality: *Does the results 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 *research 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*.

Resonance: *Does the theory generated resonate among participants/informants?* – While the final rounds of data collection was shut down prematurely on account on the pandemic situation, we were

[4]Technically, we would, for (obvious) visibility and (equally obvious) safety reasons, prefer open plains rather than the jungle for observational studies. We will however not push the elephant metaphor further beyond the casual observation that while novice software developers might charge, they are far less likely to kill you....

fortunate to gain access to four of students (roughly 10 percent of the population) whom participated in the study. These accepted to join a small follow up session in order to allow us to gauge the resonance. This session was held in a focus group format, and the students had prior to this: a) participated a second time in the course this time acting as 'coaches', and b: read a previous version of this paper including the theory. They found the paper and the theory to be a sound description of what had transpired, and noted that the measures taken by the teachers in order to change the course was very beneficial for the students.

We take a pragmatist (Bryant, 2017) 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, but we are studying the sometimes fairly gritty surface between limits of the human mind and software development tooling through means of qualitative inquiry.

With that 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. So, just as with Bryant (2017) we want to close the door on relativisation.

8. Discussion and future work

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 previous identification of the temporal perspective (Helgesson et al., 2019), 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 and loud ambience was lamented on, the work situation was described 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 perspective for observing and analysing software development in agile teams, and it is further interesting to note the reflections of *history enriched objects* and the corresponding *temporal* cognitive dimension made by Hollan et al. (2000) alongside our findings on version control and merge operations. Future theory building and theorization based will include constructs describing distributed software development as a 'distributed cognitive production flow' and further explore the observed synthetical nature of merge operations.

In regards to our second research goal, to chart what *differences* or *similarities* that can be observed between the different group compositions, we noted that there were indeed observable, yet subtle differences between the different groups. With that said, during the field work we realised that the *differences* we could observe, to us, were significantly less interesting than the *similarities* we could observe. As a consequence we choose to use these similarities to strengthen the internal validity of our findings.

Following the 2020 iteration of the course during which these observations were made, the teachers working in the course had a few discussions on suitable interventions that could be extracted from the course. We added a more thorough introduction to Git and some harder actual hands on exercises as preparations for the 2021 course iteration. We further introduced more tool support to the students.

While the Covid-19 situation has forced us to teach the course via Zoom and arguably made the whole course (that depends on teamwork) considerably more difficult we systematically noted that the students were suffering less from version control issues and were actually appearing to be more productive than previous years. For obvious reasons the pandemic situation prevented us from doing a more thorough follow up in the field.

Acknowledgement

The authors wish to thank all the participating students for their invaluable contributions, as well as course responsibles for providing the opportunity to conduct the study. We further thank Softhouse[5] for providing time for the second author and the PPIG reviewers, and audience, for valuable feedback.. The work described in this paper was conducted in the ELLIIT[6] strategic research environment.

9. References

Abend, G. (2008, June). The Meaning of 'Theory'. *Sociological Theory*, *26*(2), 173–199. Retrieved 2021-05-19, from https://doi.org/10.1111/j.1467-9558.2008.00324.x (Publisher: SAGE Publications Inc) doi: 10.1111/j.1467-9558.2008.00324.x

Beck, K. (1999). *Extreme programming explained: embrace change*. USA: Addison-Wesley Longman Publishing Co., Inc.

Begum, M. (2021). Cognition and Distributed Cognition in Software Engineering Research[WIP]. In (p. 8). Psychology in Programming Interest Group.

Bertelsen, O. (1997, November). Toward A Unified Field Of SE Research And Practice. *IEEE Software*, *14*(6), 87–88. doi: 10.1109/MS.1997.636682

Blackwell, A. F., Petre, M., & Church, L. (2019, November). Fifty years of the psychology of programming. *International Journal of Human-Computer Studies*, *131*, 52–63. Retrieved 2019-12-02, from http://www.sciencedirect.com/science/article/pii/S1071581919300795 doi: 10.1016/j.ijhcs.2019.06.009

Bryant, A. (2017). *Grounded Theory and Grounded Theorizing – Pragmatisism in Research Practice*. Oxford, UK: Oxford University Press.

Buchan, J., Zowghi, D., & Bano, M. (2020). Applying Distributed Cognition Theory to Agile Requirements Engineering. In N. Madhavji, L. Pasquale, A. Ferrari, & S. Gnesi (Eds.), *Requirements Engineering: Foundation for Software Quality* (pp. 186–202). Cham: Springer International Publishing. doi: 10.1007/978-3-030-44429-7_14

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

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

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

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

Church, L., Soderberg, E., & Elango, E. (2014, June). A case of computational thinking: The subtle effect of hidden dependencies on the user experience of version control. In B. du Boulay & J. Good (Eds.), *Proceedings of psychology of programming interest group annual conference* (p. 123-128). Brighton, United Kingdom.

De Rosso, S. P., & Jackson, D. (2016). 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* (pp. 292–310). New York, NY, USA: ACM. doi: 10.1145/2983990.2984018

Flor, N. V., & Hutchins, E. L. (1991). Analyzing distributed cognition in software teams: a case study of team programming during perfective maintenance. In J. Koenemann-Belliveau, T. G. Moher,

[5] https://www.softhouse.se

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

& S. P. Robertson (Eds.), *Proceedings of Empirical Studies of Programmers* (p. 36-64). Norwood, NJ, USA: Ablex Publishing Corporation.

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

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

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

Hansen, S., & Lyytinen, K. (2009, August). Distributed Cognition in the Management of Design Requirementsdistributed cognition in the management of design requirements. In R. C. Nickerson & R. Sharda (Eds.), *Proceedings of the 15th americas conference on information systems* (p. 266). San Francisco, California, USA.

Hedin, G., Bendix, L., & Magnusson, B. (2005, January). Teaching extreme programming to large groups of students. *Journal of Systems and Software*, *74*(2), 133–146. doi: 10.1016/j.jss.2003.09.026

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

Helgesson, D., & Runeson, P. (2021). Towards grounded theory perspectives of cognitive load in software engineering. In *Ppig 2021*.

Henrich, J., Heine, S. J., & Norenzayan, A. (2010, June). The weirdest people in the world? *Behavioral and Brain Sciences*, *33*(2-3), 61–83. doi: 10.1017/S0140525X0999152X

Hollan, J., Hutchins, E., & Kirsh, D. (2000, June). Distributed Cognition: Toward a New Foundation for Human-computer Interaction Research. *ACM Trans. Comput.-Hum. Interact.*, *7*(2), 174–196. doi: 10.1145/353485.353487

Höst, M., Regnell, B., & Wohlin, C. (2000, November). Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering*, *5*(3), 201–214. doi: 10.1023/A:1026586415054

Höst, M., Wohlin, C., & Thelin, T. (2005, May). Experimental context classification: incentives and experience of subjects. In *Proceedings of the 27th international conference on Software engineering* (pp. 470–478). St. Louis, MO, USA: Association for Computing Machinery. doi: 10.1145/1062455.1062539

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

Lenberg, P., Feldt, R., & Wallgren, L. G. (2015, September). Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, *107*, 15–37. doi: 10.1016/j.jss.2015.04.084

Liebel, G., & Chakraborty, S. (2021, March). Ethical issues in empirical studies using student subjects: Re-visiting practices and perceptions. *Empirical Software Engineering*, *26*(3), 40. Retrieved 2021-05-10, from `https://doi.org/10.1007/s10664-021-09958-4` doi: 10.1007/s10664-021-09958-4

Mangalaraj, G., Nerur, S., Mahapatra, R., & Price, K. H. (2014, March). Distributed Cognition in Software Design: An Experimental Investigation of the Role of Design Patterns and Collaboration. *MIS Quarterly*, *38*(1), 249–A5.

Miller, G. A. (1956). The magical number seven plus or minus two: some limits on our capacity for processing information. *Psychological review*, *63*(2), 81–97. doi: 10.1037/h0043158

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

Perez De Rosso, S., & Jackson, D. (2013). 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* (pp. 37–52). New York, NY, USA: ACM. doi: 10.1145/

2509578.2509584

Ramasubbu, N., Kemerer, C. F., & Hong, J. (2012, September). Structural Complexity and Programmer Team Strategy: An Experimental Test. *IEEE Transactions on Software Engineering*, *38*(5), 1054–1068. doi: 10.1109/TSE.2011.88

Rigby, P. C., & Thompson, S. (2005, October). Study of novice programmers using Eclipse and Gild. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange* (pp. 105–109). San Diego, California: Association for Computing Machinery. doi: 10.1145/1117696.1117718

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

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

Shaft, T. M., & Vessey, I. (2006). The Role of Cognitive Fit in the Relationship between Software Comprehension and Modification. *MIS Quarterly*, *30*(1), 29–55. doi: 10.2307/25148716

Sharp, H., Dittrich, Y., & de Souza, C. R. B. (2016, August). The Role of Ethnographic Studies in Empirical Software Engineering. *IEEE Transactions on Software Engineering*, *42*(8), 786–804. doi: 10.1109/TSE.2016.2519887

Sharp, H., Giuffrida, R., & Melnik, G. (2012, May). Information Flow within a Dispersed Agile Team: A Distributed Cognition Perspective. In *Agile Processes in Software Engineering and Extreme Programming* (pp. 62–76). Springer, Berlin, Heidelberg. doi: 10.1007/978-3-642-30350-0_5

Sharp, H., & Robinson, H. (2004, December). An Ethnographic Study of XP Practice. *Empirical Software Engineering*, *9*(4), 353–375. doi: 10.1023/B:EMSE.0000039884.79385.54

Sharp, H., & Robinson, H. (2006, June). A Distributed Cognition Account of Mature XP Teams. In *Extreme Programming and Agile Processes in Software Engineering* (pp. 1–10). Springer, Berlin, Heidelberg. doi: 10.1007/11774129_1

Sharp, H., & Robinson, H. (2008, July). Collaboration and co-ordination in mature eXtreme programming teams. *International Journal of Human-Computer Studies*, *66*(7), 506–518. doi: 10.1016/j.ijhcs.2007.10.004

Sharp, H., Robinson, H., & Petre, M. (2009, January). The role of physical artefacts in agile software development: Two complementary perspectives. *Interacting with Computers*, *21*(1-2), 108–116. doi: 10.1016/j.intcom.2008.10.006

Sharp, H., Robinson, H., Segal, J., & Furniss, D. (2006, July). The role of story cards and the wall in XP teams: a distributed cognition perspective. In *AGILE 2006 (AGILE'06)* (pp. 11 pp.–75). doi: 10.1109/AGILE.2006.56

Stol, K.-J., & Fitzgerald, B. (2018, September). The ABC of Software Engineering Research. *ACM Trans. Softw. Eng. Methodol.*, *27*(3), 11:1–11:51. doi: 10.1145/3241743

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

Storey, M.-A., Damian, D., Michaud, J., Myers, D., Mindel, M., German, D., ... Hargreaves, E. (2003, October). Improving the usability of Eclipse for novice programmers. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange* (pp. 35–39). Anaheim, California: Association for Computing Machinery. doi: 10.1145/965660.965668

Svahnberg, M., Aurum, A., & Wohlin, C. (2008). Using Students As Subjects - an Empirical Evaluation. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (pp. 288–290). New York, NY, USA: ACM. doi: 10.1145/1414004.1414055

Thornberg, R., & Charmaz, K. (2014). Grounded theory and theoretical coding. In *The SAGE Handbook of qualitative data analysis.* London, UK: SAGE Publications.

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

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

Walenstein, A. (2002). *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework* (Unpublished doctoral dissertation). School of Computing Science, Simon Fraser University.

Zaina, L. A. M., Sharp, H., & Barroca, L. (2021, March). UX information in the daily work of an agile team: A distributed cognition analysis. *International Journal of Human-Computer Studies*, *147*, 102574. Retrieved 2021-03-04, from https://www.sciencedirect.com/science/article/pii/S1071581920301762 doi: 10.1016/j.ijhcs.2020.102574