How do Students Experience and Judge Software Comprehension Techniques?

Regina Hebig
Truong Ho-Quang, Rodi Jolak, Jan Schröder
hebig@chalmers.se,truongh@chalmers.se
rodi.jolak@gu.se,jan.schroder@gu.se
Chalmers | University of Gothenburg
Gothenburg, Sweden

Humberto Linero
hlinero18@gmail.com
University of Gothenburg
Gothenburg, Sweden

Magnus Ågren, Salome Honest Maro
magnus.agren@chalmers.se
salome.maro@cse.gu.se
Chalmers | University of Gothenburg
Gothenburg, Sweden

ABSTRACT

Today, there is a wide range of techniques to support software comprehension. However, we do not fully understand yet what techniques really help novices, to comprehend a software system. In this paper, we present a master level project course on software evolution, which has a large focus on software comprehension. We collected data about student's experience with diverse comprehension techniques during focus group discussions over the course of two years. Our results indicate that systematic code reading can be supported by additional techniques to guiding reading efforts. Most techniques are considered valuable for gaining an overview and some techniques are judged to be helpful only in later stages of software comprehension efforts.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Documentation**.

KEYWORDS

Program comprehension, education

ACM Reference Format:
Regina Hebig, Truong Ho-Quang, Rodi Jolak, Jan Schröder, Humberto Linero, Magnus Ågren,, and Salome Honest Maro. 2020. How do Students Experience and Judge Software Comprehension Techniques? . In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387904.3389283

1 INTRODUCTION

Software comprehension is estimated to consume 58% of software developers' time [32]. A lot of research has been done on creating curricula for educating students in software comprehension, maintenance, and evolution. For example, the IEEE-CS/ACM Initiative *Computing Curricula: Software Engineering* (CCSE) developed some recommendations for software evolution curricula[1], which include program comprehension and reverse engineering as essential aspects. The work is based on panel discussions, e.g. the one organized by Van Deursen et al. [30] at the ICSM'02 conference.

Follow-up studies show that the focus on comprehension of software systems is still relatively small [30]. For example, AbuLail and Shkoukani [1] analysed the prevalence of reverse engineering education in Jordanian Universities and conclude that there is a lack of relevant education. Also Bordin and Benitti [3] found that education involving legacy-system related topics are addressed too little in the example of Brazilian universities. Thus, there still is a need for more guidance on how to teach software comprehension.

To teach software comprehension, educators have to select between a multitude of different *comprehension techniques*. This includes strategies, e.g. systematic code reading, as well as a large range of technologies, e.g. for documentation generation, software visualization, or automated reverse engineering.

However, we do not fully understand yet what techniques help novices[2] to comprehend and familiarize themselves with a legacy software system. This lack of knowledge impacts the effectiveness of our teaching. In addition, confronting students with techniques that do not benefit them yields the risk that they might not use or try out new comprehension techniques after their education ends. In consequence, the distribution and acceptance of new effective software comprehension techniques might be hindered. To address this issue, we formulate the following research question: *How do students experience and judge software comprehension techniques?*

In this paper, we present a project course on software evolution, which has a large focus on software comprehension. The overall course design is partially inspired by the CCSE curriculum for software evolution. The course is open to master students in their second year only and includes project work on a real legacy system. During the years 2018 and 2019, we performed focus group sessions at the end of the initial comprehension phase after five weeks. In a semi-structured discussion, we asked students to reflect about their experience with different comprehension techniques learned during the course and asked them to judge what techniques they would use in future projects. We analyze notes made during 20 of these focus group sessions, including altogether 97 students. The aim of

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389283

[1]CCSE Initiative - Teaching Software Evolution http://www.program-transformation.org/Transform/TeachingSoftwareEvolution

[2]Note, that many studies argue that master's students act and decide in a comparable manner as professionals [27]. We use the term novices here to refer to developers who are new to the software system they are working on.

the analysis was to get a qualitative and exploratory understanding of students' perception of software comprehension technologies.

Our results indicate that systematic code reading can be supported by other technologies to guiding reading efforts. Furthermore, most of the studied comprehension technologies are considered either valuable for gaining a sense of overview, or are judged to be helpful only in later stages of software comprehension efforts.

The paper is structured as follows: Section 2 provides an overview about related work. In Section 3 we introduce the design of the comprehension-related parts of our software evolution course. In Section 4 we present our method and in Section 5 results are summarized. We discuss lessons learned and implications of the results in Section 6. Finally we conclude in Section 7.

2 RELATED WORK

In this section, we discuss related work on teaching and course designs for evolution and software comprehension, works about comprehension tools for programming education, as well as empirical studies on software comprehension.

2.1 Teaching software comprehension

Course designs to cover comprehension. In the early 2000s a series of lecturers published their course designs for teaching software evolution and maintenance. For example, Austin and Samadzadeh [2] propose a course design for a junior level class which centers around a couple of lectures, e.g. on metrics, and a large scale project on maintenance with major focus on code comprehension.

Petrenko et al. [20] present a course design for open source software evolution, where teams of 4 to 6 students are asked to evolve an open source system during three graded phases. PhD students serve as supervisors and customers in this course and meet the students weekly. Finally, Tremblay et al. [29] present a course design that is based on a "Maintenance Corpus", i.e. a baseline software that can be reused for maintenance tasks during the course. The software was initially developed by an old student team and further adjusted by the teachers.

Course designs and teachers perspective. Some educators present their course designs together with a discussion of lessons learned from the teacher's perspective. For example, El-Ramly [9] presents a master-level course design and lessons learned from a maintenance and re-engineering course. He concludes that, at the time, there was a shortage of educational materials and pedagogical research on the topic. Also the selection of the right tools was considered a challenge, as theoretical knowledge should be accompanied by practical experience, but the number of tools should also not be too high. In addition, El-Ramly points out the need to balance between toy examples, which are not suitable to teach practical aspects, and real systems, which might be infeasible to introduce in a class-room.

Pierce [21] describes experiences from an undergraduate course. He used a system to maintain (of around 4000 lines of code), rather than a system to build and reports how this helps to illustrate to students the need for a process, change management, readable code and good documentation. Koskinen [15][16] presents an empirical study to evaluate a seminar-based course design for software maintenance and evolution. They found among other things that success in the seminar predicted the student's success in a later software engineering course examination.

Course designs and student's perspective. Other educators focused on collecting the perspective of students to evaluate their course designs. For example, Postema et al. [22] present a course to teach corrective, adaptive, perfective, and preventative maintenance activities to second year students. The used system was about 2000 source lines of code and the course had no specific focus on comprehension. Students' feedback is summarized as positive. Nascimento et al. [17] discuss a list of reference curricula for software evolution, including seminars and projects. They compare these with their own course, which is a seminar-based course where students read research papers. Students appreciated that they gained a broad view of the maintenance and evolution area. Pérez-Castillo et al. [19] present and evaluate a 3 hour teaching unit for software reengineering, consisting of a theoretical part and a practice session. They summarize several lessons learned, including the observation that the biggest challenge was the integration of a new functionality. They also found that students appreciated reverse engineering tools, but required detailed explanations. Szabo [28] present a teaching unit within a project-based educational set-up, which focuses on maintenance using old code produced by other students. The system given to the students included no documentation and students were asked to adapt the system. Students identified the lack of comments and naming conventions as a hindrance to comprehension.

Summary. There is today a good amount of literature that can be used as inspiration when designing software evolution and maintenance courses. However, so far only few course designs focus explicitly on what comprehension techniques are understood and considered useful by students. We argue that this is crucial. The goal of higher education should be to equip students with the mindset and ability to life-long build, maintain, and extend a portfolio of techniques that they can use for comprehension. For that it is crucial that they experience and understand the purposes and benefits of techniques taught in university.

2.2 Comprehension tools for education

Some comprehension tools were introduced with the explicit focus to improve programming education. For example, Čisa et al. [5, 6] used a program called Jeliot 3 that presents visualizations of applications, showing method calls and operations. The tool can also animate the program execution. They analyzed test results of 400 students and found that the tool helped students to achieve better results. Sulaiman et al. [26] proposed the TubVis tool to illustrate analysis results over the source code. They argue that lightweight tools are preferable for educational purposes.

2.3 Empirical studies on comprehension

In 2007 Di Penta et al. [8] call for more empirical studies on program comprehension. In 2017 Schröter et al. [23] followed up and presented results from a literature review on papers published at the International Conference on Program Comprehension. They found that by far most papers study comprehension of source code and program behavior. A smaller group of papers focuses on comprehension of testing, APIs, requirements, or documentation. Case studies and experiments are the most common methods used.

Examples of empirical studies on software comprehension include studies with students. For example, Karahasanović and Thomas [14] performed a controlled experiment on student's ability to maintain object-oriented programs. They had 34 participants, who studied computer science in their third year. The participants were asked to perform three maintenance tasks during seven hours on a Java system that had 26 classes and around 3600 lines of code. The main identified difficulties concern the ability to understand the program logic, algorithms, change impacts, and inheritance of functionality. Other examples focus on specific comprehension techniques. For example, Wettel et al. [31] performed a controlled experiment with 41 participants, showing that code cities can help to decrease task completion time and increase correctness of typical software comprehension tasks. Another example is presented by Cornelissen et al. [7] who compare trace visualization techniques in an experiment in terms of task completion time.

Bragdon et al. [4] and Jolak et al. [12] presented CodeBubbles and OctoBubbles, which are interactive environments to concurrently visualize and edit different code pieces and models. Evaluations of both approaches indicate benefits for comprehension, due to the reduced need to navigate between different windows.

To summarize, there is already a large base of empirical studies about different software comprehension techniques. We aim to add to this taking in the perspective of novices at a very early stage of software comprehension, who have a portfolio of different techniques available.

3 EDUCATIONAL SETUP

The 15 credits project course on software evolution is mandatory for students at the start of the second and final year of their master education. Between 36 and 60 students participate in the course each year. They work in teams of normally 4 to 5 students.

3.1 Educational Philosophy

The idea for this project course is based on three closely connected learning theories: constructivism (including the subcategory "project-based learning" [13]), connectivism [24], and social constructivism [18]. Constructivism promotes the idea that students learn by active exploration and inquiry, constructing knowledge by working with a subject [13]. Project courses create an environment for this form of learning, exposing students to large problems that require step-wise exploration of and experimentation with alternative solutions. In software engineering this often goes hand in hand with connecting students to online resources, encouraging them to explore content created by others to learn about technologies and to share their own contents as well, which follows the theory of Connectivism [24]. We consider this connection to online resources and forums crucial, as students will maintain and learn in these communities long after they have left university. Finally, students are working in teams during these courses, which should prepare them for the later reality in companies, where software developers rarely work alone. The teamwork in project courses is also motivated by the idea of social constructivism [18]: by being exposed to peers' understanding of problems and solutions, we hope that each individual student reaches a deeper learning about the subject and trains their ability to try and compare their own and other's ideas.

3.2 Course Design

The course as a whole focuses on software evolution. Of the 16 weeks, the first five are dedicated to software comprehension (roughly a third of the course). This part of the course is of the main interest for this study. The course part is aiming to fulfill the following excerpt of the course's official learning outcomes.

- summarize the state of the art in methods and tools for software evolution tasks, such as program comprehension and software refactoring,
- discuss the challenges associated with software evolution,
- extract a software product's architecture from a given code base and evaluate the quality of the software product,
- detect and judge needs for quality improvement or evolution in an authentic software product

The project is accompanied by a set of lectures, individual assignments, as well as supervision sessions and milestone meetings. Figure 1 shows the typical schedule for the first half of the course.

There are four lectures (sessions) during the first half of the course: the course introduction, in which the course structure and projects are presented, a session on program comprehension techniques, a session on clone detection, and a session of refactoring.

The practical part of the project involves comprehending and evolving a legacy software system. In the first five weeks, in parallel to the lectures, the focus for the students is on comprehending the system and requirements for the project. Students receive an installation guide from day one. The guide covers both, the legacy system and tools used for comprehension and system analysis during the course. At the end of week one we have an installation session. During that time all supervisors are available to help students who have installation problems. Note that the goal of this session is to help each team to have at least one running version of the system.

The project has 5 group milestones (GMs), to check in regularly with the team's progress. The first GM is located at the end of week 5 and is focused on assessing whether the teams have reached a first level of understanding of the system. All further milestones have two weeks of time between each other. Goal for these milestones are planned at the end of each previous milestone by the team together with their supervisor. Groups collect up to one point for each successfully passed milestone.

The three individual milestones are assignments that are performed on the project system, but by the team members individually. This way we want to promote that all team members participate in comprehension and refactoring-related tasks. All students are encouraged to perform these tasks and can collect this way points for the final grade. Apart from that, the teams meet their supervisor twice a week up to 30 minutes. They also have large project work rooms available during these mornings, which they can use to meet and interact and exchange with other teams in the project. Also, students were strongly encouraged to meet and interact with the project's "customers" during that time.

3.3 Software Comprehension

The first topic-lecture of the course is an introduction to software comprehension techniques. The slide shown in Figure 2 is used to provide an overview to diverse techniques to the students as well

Week	Mo.	Tuesday	We.	Th.	Friday
1		09:00-12:00 Session: Course Introduction and Project Presentation			09:00-12:00 Installation session
2		08:00-10:00 Session: Program Comprehension			08:00-12:00 Supervision Meetings
3		08:00-12:00 Supervision Meetings *24:00 Individual Milestone: Comprehension*			08:00-12:00 Supervision Meetings
4		08:00-10:00 Session: Clone Detection			08:00-12:00 Supervision Meetings
5		08:00-12:00 Supervision Meetings *24:00 Individual Milestone: Clone Detection*			08:00-12:00 Supervision Meetings *(Evaluation GM 1, Planning GM 2, & Reflection session Program Comprehension)*
6		08:00-12:00 Session: Refactoring			08:00-12:00 Supervision Meetings
7		08:00-12:00 Supervision Meetings *24:00 Individual Milestone: Refactoring*			08:00-12:00 Supervision Meetings *(Evaluation GM 2 & Planning GM 3)*

Figure 1: Typical schedule of the course's starting phase (the first half of the semester).

as provide a structure for this lecture. Techniques illustrated in blue are the ones that are taught in detail during the lecture, namely:

- Systematic Code Reading
- Documentation Generation (introduced using Doxygen[3])
- Reverse Engineering (introduced using the tools ObjectAid[4] and Doxygen in combination with GraphViz[5])
- Software Visualization (introduced using SonarQube's[6] Tree Maps and Code Cities[7])

The lecture is active, including exercises on very small systems during the lecture time. Therefore, we actively use only tools that are stable and run on most machines and operating systems. Unfortunately, that excludes some state-of-the-art techniques from being mandatory content. However, students are of course informed that better, state-of-the-art tools exist and can use these in their projects, if they want to and have access.

We aim at teaching our students to a critically reflect about these techniques. In the assignment (individual milestone) on comprehension students are asked to generate a documentation of the project's systems as well as to create two class diagrams on different abstraction levels using reverse engineering techniques. They are ask to argue for their abstraction choices. Finally, group milestone 1 ends with a reflection session on the experiences that the students had with software comprehension during the first 5 weeks.

3.4 The Projects

Over the years, we tried out different project set-ups in this course, ranging from open source projects to projects with external partners. Each set-up has advantages and disadvantages. In general, we experienced that students are more motivated and their work on the project is more targeted, when there is a customer who provides a task or requirements for the evolution project.

We argue that this form of motivation and target is crucial. We can expect students to act differently during a software comprehension exercise, if they have a real need to gain an understanding of the system to evolve it further compared to a situation where the single goal is to pass an assessment of comprehension in form

[3]Doxygen http://www.doxygen.nl
[4]ObjectAid http://www.objectaid.com/home
[5]GraphViz http://www.graphviz.org/
[6]SonarQube https://www.sonarqube.org/
[7]Softvis3d plugin for SonarQube http://softvis3d.com/#/

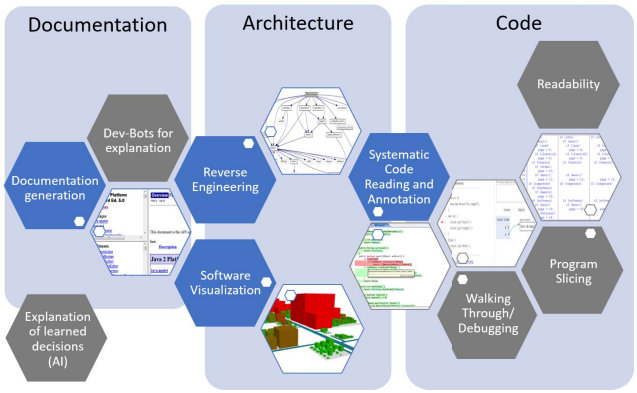


Figure 2: Overview slide about comprehension techniques discussed and mentioned during the software comprehension lecture.

of questions about the system. On the other hand, using external project partners has a risk as we as teachers have less control about the amount of time they are putting into interacting with the students.

2018. In the course instance in 2018, we worked with such an external partner. Thus, the students had access to customers who had limited knowledge of programming. The customers provided a system that they worked with, together with requirements on how to evolve it. In addition, the customer invited the students to a factory visit at the start of the weeks three and five of the project, to explain the domain and allow students to ask questions about the requirements of the project. The customer's domain was additive manufacturing. The system given to the students provided simple tracing functionality, allowing the customer to store and analyze diverse files, images, and measurements created during the manufacturing process. The system consisted of three parts, one back-end and two front-ends. The back-end was build using C# and hosted as an Amazon service (size around 230 files, including used packages). The first front-end was a website built using Angular and typescript (size around 60 files) and the second front-end was build as android application (size around 140 files).

2019. For the course instance in 2019, we decided to work with "internal" stakeholder represented by PhD students. The used system was the McGill's Qualizer[8], which is an Eclipse-based open source systems to code interview data for qualitative studies. The PhD students formulated wishes for the new system, e.g. support for playing additional audio files or advanced coding mechanisms allowing for coded text pieces to overlap. The students could contact and meet these customers throughout the course to learn about domain and requirements or to present project progress and get feedback on early drafts and prototypes. The system consists of around 320 files (Java) and was last released in 2012.

4 METHOD

To address the major research question on the students' experience and judgment of software comprehension techniques, we perform an exploratory study using focus group discussions. We expose

[8]Qualizer https://qualyzer.bitbucket.io/

teams of students to the experience of having to comprehend a software systems to later evolve it, providing them with a set of tools that they can use. We then used focus group discussions to collect qualitative data about the participant's experience.

4.1 Participants

All participants were in their final year of master education. All students study in an international program, located in Sweden. While half of the students are Swedish, the other half are students from all around the world. Of the 22 teams participating in the focus groups, 20 agreed to their data being used. This included 13 teams in the course instance from 2018 and seven teams in the course instance from 2019. Most teams consisted of five students, with exception of two teams, of which one consisted of four and one of three students. Thus, altogether 97 students participated in the study. Of these, 82 participants were male and 15 were female. Seven teams were mixed (male and female students), one team consisted of only female students, and 12 teams consisted of only male students.

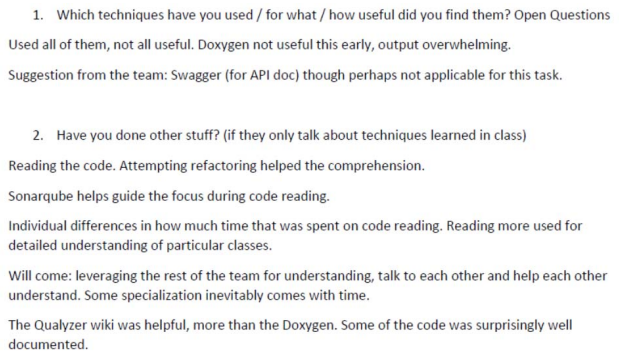
4.2 Data collection

Data was collected during two instances of the course running in the years 2018 and 2019. We used the reflection sessions from the course as natural focus groups for the data collection. These sessions are part of wrapping up the first big milestone of the course after the students had 5 weeks to start comprehending the system.

While the whole session takes 45 minutes per team, we only collected data during the last 15 minutes. The first 20 minutes were used by the teams to demonstrate that they indeed managed to gain an understanding of the system, its structure, its functionality, and roles of different packages and classes. During this part of the session, students collect their first points counting towards the final grade. The points are given live during the session to the students. Note that this is rather a check-point than a obstacle/test, where most teams achieve full points. After that, the teams spent 10 minutes to plan their goals for the next milestone, which will be two weeks from that point in time. Finally, the last 15 minutes, we encourage to students to reflect on what techniques and approaches have actually helped them reaching their current understanding of the software system. During these last 15 minutes we collected data. Each of these focus group sessions was run by the teaching assistant (a Phd student) responsible for the supervision of the team.

4.2.1 Discussion guide. To make sure that all reflection sessions were comparable and also to help the students to have a systematic approach to their discussions, we developed a discussion guide. This guide was used by the teaching assistants to moderate the discussions. The discussion guide included as a reminder the list of comprehension techniques taught during the lecture at the start of week two. In addition, the discussion guide included four questions:

(1) Which techniques have you used / for what / how useful did you find them?
(2) Have you done other things?
(3) Did you not use (roles, sonarqube, code cities, ..) and why?
(4) Anything you would do differently/techniques you would use again/ differently next time?


1. Which techniques have you used / for what / how useful did you find them? Open Questions

Used all of them, not all useful. Doxygen not useful this early, output overwhelming.

Suggestion from the team: Swagger (for API doc) though perhaps not applicable for this task.

2. Have you done other stuff? (if they only talk about techniques learned in class)

Reading the code. Attempting refactoring helped the comprehension.

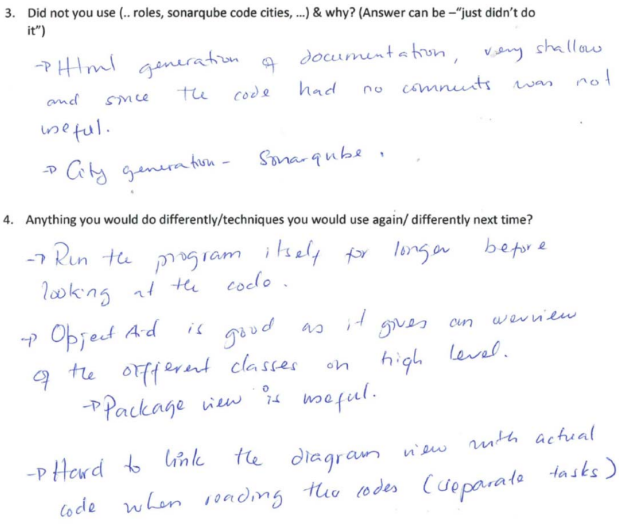
Sonarqube helps guide the focus during code reading.

Individual differences in how much time that was spent on code reading. Reading more used for detailed understanding of particular classes.

Will come: leveraging the rest of the team for understanding, talk to each other and help each other understand. Some specialization inevitably comes with time.

The Qualyzer wiki was helpful, more than the Doxygen. Some of the code was surprisingly well documented.

Figure 3: Example notes from a focus group session (digital).


3. Did not you use (.. roles, sonarqube code cities, ...) & why? (Answer can be –"just didn't do it")

→ Html generation of documentation, very shallow and since the code had no comments was not useful.

→ City generation - Sonarqube.

4. Anything you would do differently/techniques you would use again/ differently next time?

→ Run the program itself for longer before looking at the code.

→ Object Aid is good as it gives an overview of the different classes on high level.
→ Package view is useful.

→ Hard to link the diagram view with actual code when reading the codes (separate tasks).

Figure 4: Example notes from a focus group session (handwritten).

4.2.2 Form of collected data. Due to the embedding in the educational set-up, we decided against asking students whether we could make audio-recordings during the sessions. One of the reasons is that such records would have created data that can be traced to individuals. We were worried that this would disturb the participants and make them participate less in the focus group.

Instead, we decided to make notes during the focus group sessions, using the question guide as a form for that. This was done by some of us digitally and by some with pen on paper. The Figures 3 and 4 show two examples of these notes. Notes were made by one to two of us teachers: the teaching assistant responsible for a team and in half of the cases also the main course responsible.

4.2.3 Consent. We performed the focus group meetings as part of the educational training (reflection sessions). Thus, they were run with all teams, independent on whether a teams agreed to their data being used or not. One important step for us was to encourage the teams to be critical and ensure them that their answers and participation do not count during the grading of the course in any way. We asked the teams whether they agree with us using the data pointing out again that the answer does not affect their grading. We

informed the students about the following. The data would be used anonymously and in an aggregated form, only, i.e. no notes were made that would connect statements to individual students. Also, the data would be used for an educational publication, to allow teachers at other Universities to benefit from the insights.

4.3 Analysis

To analyze the data we used systematic coding. We stored all meeting notes in a large excel sheet, sorted by groups and questions. We coded the data in a first iteration, using codes such as "SonarQube", "Code Reading", or "Focus". Note that we partially assigned multiple codes per note. In a second step, we created cards for each coded note. Each card included the note itself, the code, the id of the focus group and the id of the question to which the answer belonged. Note that this was necessary, as we wanted to distinguish answers to the questions about what was not used and what techniques the students would use again in future. Maintaining the ids allowed us to assess the number of focus groups discussing each theme.

We then sorted cards into themes based on the initially assigned codes. During that step some cards were moved between the stacks, as they turned out to better fit to other themes, several themes were merged, and some themes renamed. Some of the themes directly match comprehension techniques, while others touch additional aspects regarding comprehension in general. A set of remaining cards were sorted into the theme "other". As a results we gained 12 themes (without "other"). For each theme we then sorted the cards into sub-groups, to allow for a detailed discussion of the findings.

Note that, although we counted the number of focus groups who discussed each theme, we do not claim that this can be used for quantification of importance. First of all, we would require more data to establish statements about frequency. The second problem is that we specifically asked for those techniques that we also taught during the course. As a consequence, these techniques naturally have a higher number of teams talking about them when other approaches/themes. We nonetheless decided to present these numbers, to allow the reader to assess validity of our results.

5 RESULTS

In the following, we present the results from the 20 focus group discussions. The results are sorted into 12 themes, of which the biggest sub-group are themes about technologies for comprehension. For this sub-group of themes, we present positive aspects, negative aspects, and, if mentioned, possible adjustments. As illustrated in Figure 5, each of the themes was discussed by at least 4 teams.

5.1 Existing Documentations and Wikis

Nine teams discussed pre-existing documentations and wikis. In general wikis, including sections on frequently asked questions, and existing documentation were considered useful. Two teams had the feeling that they did not make enough use of existing documentation, yet. However, four teams also discussed the lack of documentation in the given systems, mentioning the wish for more comments and diagrams. Finally, one team mentioned that they found inconsistencies between documentation and code.

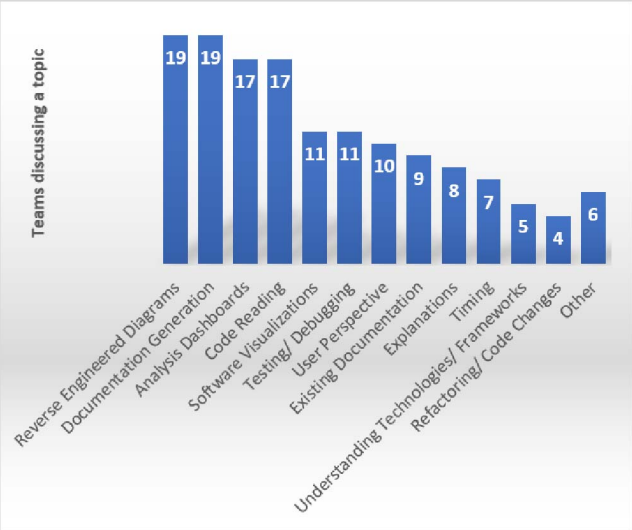


Figure 5: Number of focus groups discussing a theme.

5.2 Documentation Generation

Documentation generation was used by most teams early in the project, encouraged by an individual assignment. During the focus group sessions, 19 of the 20 teams discussed their experience with this technology. The tools used for documentation generation were Doxygen and Compodoc[9]. The main hindrance for benefiting from documentation generation is the lack of useful comments and meaningful naming in the code. Five teams discussed this issue during their focus group sessions. Students in seven of the focus group sessions reported that documentation generation helped them during the comprehension. This help mostly manifested in providing an overview of the system and a starting point for further investigations. However, during ten teams' sessions, documentation generation was reported to be "cool", but not useful for comprehension. Note, that there were two teams that included students who considered documentation generation to be useful as well as students you found it not useless. In the end, students from three teams expressed that they would not use documentation generation again. In contrast, students from four teams said they would use documentation generation again, if there were better comments in the code and likely later during the comprehension process.

5.3 Analysis Dashboards - SonarQube

17 of the teams discussed the impact of using SonarQube on software comprehension. In eleven teams, students reported the use of SonarQube to be beneficial for software comprehension. Named benefits were, to get a quick overview about the system in terms of numbers and statistics, to identify problems, to identify the biggest parts of the system, as well as to identify the most "risky" parts. One team anecdotally reported that SonarQube helped them to understand that the back-end of the analyzed system did not contain any domain knowledge. Furthermore, some teams considered SonarQube's quality analysis to be useful for comprehension. Thereof, clone detection was named most often. For example, one team

[9]Compodoc https://compodoc.app/

reported that SonarQube's clone detection showed them that handlers had a lot of duplicates. This insight helped them to further understand how modularity was achieved in the system. On the other hand, in four teams students reported that the use of SonarQube did not help with the comprehension. Students in three teams stated that they would not use SonarQube again in context of software comprehension. However, in nine teams students stated the opposite, emphasizing that they would use the tool again.

5.4 Software Visualizations

Software visualization was introduce to the students in form of Code Cities. Only two teams considered this to be useful for comprehension of the software. In contrast, seven teams stated that they would not use code cities again for software comprehension as they did not consider them to be useful. Finally, three teams discussed abstraction to be an issue of software visualization. They wished for an ability to better filter the represented data and to switch between abstraction levels.

5.5 Reverse Engineered Diagrams

Altogether 19 of the 20 teams discussed reverse engineering during the reflection sessions. Our students used a variety of tools for reverse engineering and generation of diagrams. The most often used tool was ObjectAid, which was also the main tool taught. However, many teams also tried other tools. This was partially driven by the the programming languages is the systems, as many tools are specific to one language, only. These tools were: Visual Studio Class View[10], StarUML[11], Intelij (enterprise version)[12], Visual Studio code maps[13], Compodoc, UMLet[14], as well as the diagram generation that is part of the documentation generation when combining Doxygen and GraphViz.

Twelve teams reported reverse engineered diagrams to be useful for their comprehension of the system. The reported benefits included gaining an overview of the architecture and structure of the course code, understanding workflow and data-flow of a system, and gaining an overview about dependencies and interactions. Thus, besides class diagram also other diagrams were used, such as use case diagrams, sequence diagrams, flow diagrams, call graphs, and entity relationship diagrams (ER). Furthermore, some teams reported to use generated diagrams for guidance while reading code. However, five of these teams classified the reverse engineered class diagrams as only partially useful. They discussed package views as useful to gain an insight into the domain and an overview of the system, while pointing out that that was not enough to really comprehend the software. One team suspected whether the class diagrams might become more useful for implementation-level comprehension as the project progresses.

One team made an explicitly stated that they considered reverse engineered diagrams as not useful for comprehension, pointing out that they considered the tools as too limited. Issues with tools were discussed by ten teams. Critiques about the tools included difficulties and confusion caused by a low readability of the generated diagrams, a level of abstraction that is too low, leaving too many details in the diagrams, and missing or misleading representation of relationships. Three teams expressed that they would not used Visual Studios Class Viewer again. However, four teams explicitly mentioned that they would use reverse engineering to gain class diagrams (especially with ObjectAid) again. Besides the semi-automatic approaches, students also created diagrams manually. Three teams reported to do that while reading through the source code. A fourth team maintained and regularly updated an architecture diagram of the code as they progressed with their understanding of it.

5.6 Code Reading

The role of (systematically) reading source code was discussed in 17 of the focus groups. The teams had different strategies to move through the code, e.g. by following references from class to class.

Six of the teams discussed different artifacts that they used in combination with code reading, to guide their focus and attention. This included techniques that allowed them to gain an overview first, such as SonarQube outputs, documentation, and reverse engineered diagrams. Teams would often go back-and-fourth between the code and these artifacts.

In addition, also other code-near artifacts were subject to systematic inspections by the teams. This included, the file management system (to inspect the code structure), IDE search functions, such as Eclipse's feature to follow call hierarchies, code comments, readme files, test cases, error logs, commit histories, and the system's data base. As purposes for code reading students named getting details about classes and understanding the flow of the system. Three teams reported to utilize their understanding of the user perspective on the system while reading the code, trying to trace between both. One team reported to perform code reading in groups, having all team members in the same place looking at the screen together. Finally, six teams mentioned that they would use code reading again. Students classified it as an easy, but time consuming task.

5.7 User Perspective

Half of the teams discussed the need to understand the user perspective on the system. All of these ten groups indicated that they considered it important to run the system to gain an understanding. Four teams reported that challenges to get the system running early on in the course also was a major obstacle to their understanding of the system. Strategies to work around that issue were reported by two teams, who run system-parts independent of each other, e.g. by using Postman[15] to access the back-end.

Furthermore, four teams highlighted the need to collect domain knowledge, e.g. by talking to someone familiar with the domain, such as customers. One team looked at systems similar to the one they wanted to comprehend, to better understand the domain and its features. Finally, three teams reported that they traces GUI elements down to source code, to gain a better understanding of the system.

[10] Visual Studio Class View https://docs.microsoft.com/en-us/visualstudio/ide/viewing-the-structure-of-code?view=vs-2019
[11] StarUML http://staruml.io/
[12] Intelij https://www.visual-paradigm.com/support/documents/vpuserguide/2381/2385/66580_reverseengin.html
[13] Visual Studio Code Maps https://docs.microsoft.com/en-us/visualstudio/modeling/map-dependencies-across-your-solutions?view=vs-2019
[14] UMLet https://www.umlet.com/
[15] Postman https://www.postman.com/

5.8 Testing/Debugging

The use of testing and debugging for software comprehension was discussed by nine teams. Eight of the teams used debugging tools and breakpoints and students from two further teams stated that they would use more debugging in future projects. One of the teams did not just use debugging tools, but picked concrete errors and tried to locate them as an exercise to reach comprehension. Furthermore, three teams run tests on the application and one team spend time reading existing test cases.

5.9 Understanding Technologies/Frameworks

The role of understanding the underlying technologies was emphasized in four of the focus groups by students from both years. Two teams talked about efforts to get familiar with RCP (the Eclipse Rich Client Platform, which was the underlying framework used in the project system of the course instance 2019). They used tutorials as well as build simple trial RCP systems themselves. A third team reported to perform web-searches about super classes, to improve comprehension. One team reported that the use of the angular framework in the software system helped during the comprehension phase, since they could rely on their understanding of standard angular architectures. Last, but not least, one team discussed the impact of the systems' structure on the comprehension. They concluded that a lack of good structure negatively impacts the comprehension.

5.10 Explanations

Eight of the teams discussed explanations as a means to gain an understanding of the system. In five of these teams members discussed code together and provided code explanations to each other. In some of these cases code was read as a team, while in others the team members specialized on understanding different system parts before explaining them to each other. Furthermore, in students from four teams stated that they would reach out to former developers of the system in future situations to receive explanations. One team further stated that they would aim to document the knowledge that they gained during the first weeks.

5.11 Refactoring/ Code Change

During a relatively small number of focus group sessions single students reported on a much more active strategy towards comprehension. In four teams students discussed benefits of refactoring or otherwise changing source code to understand it. Two teams reported that they did learn about the code by performing refactorings. Furthermore, students in two teams reported that they learned about the code by fixing errors. Finally, students in one team, approached the code by actively extending a class and implementing it "from view to code".

5.12 Timing

The timing of when to use what comprehension technique and technology was discussed by seven of the teams. The general message was that certain activities require some previous understanding. For example, two teams emphasized that they would not again start to read the source code before having run and used the program. Documentation generation was mentioned by one team to have been used too early in the comprehension process. They considered the output to be overwhelming at an early stage of comprehension. Similarly, three teams commented that they felt to have used reverse engineering to create diagrams too early. They propose to rather do that after reading the source code. Furthermore, one team discussed that it would be useful to update the diagrams continuously during the comprehension process. Also debugging was mentioned as a technique that requires that a certain threshold of system understanding reached. Finally, the use of SonarQube was discussed by two teams. While one team thought that they used the tool too early, the other team proposed to use the tool even earlier to locate problems within the system from the start.

5.13 Other Strategies

Finally, some teams discussed other strategies used. Mentioned strategies were for example, choosing to focus on one aspect of the system at a time, writing and answering questions for each package, locating code responsible for a feature, and identifying risky and critical system parts. One team emphasized the importance of approaching the system from both perspectives, top-down and bottom-up. Moreover, one team wrote their own script to illustrate traces between method names.

6 DISCUSSION

The results show that students have quite diverse experiences and opinions regarding the different software comprehension technologies. However, the results also provide valuable insights about strategies students developed and benefits they perceived to gain from using different technologies. In the following, we discuss lessons learned for teaching software comprehension, as well as implications for research and threats to validity of our results.

6.1 Lessons Learned for Educators

Five lessons learned for software comprehension education are:

Towards a refined software evolution curriculum. Looking back at the CCSE curriculum for software evolution with the knowledge of this study, there are a lot of aspects for which we see a strengthened need. For example, we agree that teaching program comprehension greatly benefits from being taught in context of a course that works with legacy systems. However, our results can be seen as an invitation to develop refined recommendations for teaching reverse engineering. While the majority of teams considered reverse engineered diagrams to be beneficial, we recommend educators to not limit teaching to refactoring to class diagrams. For example, diagrams indicating data flow or dependencies were also appreciated by the students. Also, metrics on code quality should be considered as a part of a software comprehension and evolution curriculum. Austin and Samadzadeh [2] used metrics as a central aspect when teaching software comprehension. Similarly, our data shows that students find strategies to benefit from the use of metrics, e.g. by using clone detection to learn about modularization of the system.

Create a real comprehension need. Other educators have argued that software engineering activities are best taught when students know that the activities product will be used later [10, 11], as that

knowledge creates real stakes. We propose to create a real comprehension need when possible, i.e. create a set-up where students have to reflect about their own comprehension needs given a larger task to extend a system. Just as Pérez-Castillo et al. [19], we observed that integrating a new piece of functionality is a big challenge and requires a deeper understanding of a system.

However, this recommendation needs to be taken with a grain of salt. Some educators teach comprehension on a junior level [2]. However, these courses often use less technologies, e.g. do not include reverse engineering, and work with exercises rather than large projects. Thus, we can at this point not propose first or second year courses to work with evolving real legacy system.

Make students aware that comprehension is a process. Our results show that different techniques are used in combination (Section 5.6) and that the timing of the use of a technique seems to play a role (Section 5.12). This indicates that comprehension is not a single task, but a process that requires the combined use of different techniques . Students should be made aware that they are not likely to perceive all techniques as similarly useful at the start of the comprehension process and in later stages. We propose to familiarize them with different 'steps' in the comprehension process as well as likely orders in which these steps could be used. For example, gaining domain knowledge as well as identifying knowledge needs and learning about frameworks and technologies can be considered initial steps in the comprehension process. Following these, further steps could be system testing and code reading guided by high-level reverse engineered diagrams.

Don't use high-quality systems. In contrast to El-Ramly [9], our set-up of a 15 credits project course allows us to work with relatively large legacy systems. Just like Pierce [21] and Szabo [28], we perceive it to be beneficial for students to be confronted with systems that lack proper documentation and comments. This way our students did learn about the need for and benefits of documentation.

Challenge: tool selection. Finally, the selection of the right tools remains a challenge. If we could, we would use more moderns tools in our course. However, similar to El-Ramly [9] we found over the years that students have a limit towards the number of tools that they can process during one course. More importantly, students are using diverse systems and not many tools work stable on different operating systems. Similarly, commercial tools are problematic as many do not offer teaching licenses. In consequence, there is a risk that students get a bad first impression of a technology, because the tool is not state of the art. We mitigate this risk by encouraging the students to actively reflect about the technologies as part of the individual assignments and during the reflection sessions.

6.2 Implications for Research

We identified several implications for future research.

Understand technology niches. We need to better understand the niches that different comprehension technologies fill. Parts of our results seem to stand in contrast to the findings of Wettel et al. [31]. While they found that code cities increase speed and quality of comprehension, our data show no such effect. One big difference in our study is that participants were exposed to multiple technologies at once. We could observe that many of the technologies seem to fill the same functional niches when it comes to software comprehension: providing an overview and guidance during code reading. Thus, it is possible that the presence of alternative technologies fulfilling the same need, overshadows the effects of single technologies. This leads to a question on how technologies can be meaningfully combined in practice.

Consider the early phase of the comprehension process. Sillito et al. [25] distinguished in their study between question asked by novices and questions asked by programmers working on systems they are already familiar with. Comparing our results with related studies, we argue that this distinction is essential as it likely explains some of the differences of our findings to related work. For example, Wettel et al. [31] defined comprehension tasks, such as "locate all the unit test classes of the system [..]" or "find the three classes with the highest number of methods in the system". These questions may likely be asked in later stages of the comprehension, e.g. when the developer is already familiar with the overall architecture and some details of the system. However, they are likely not relevant for a very early stage of comprehension. To draw reliable conclusions on the early-comprehension phase, as well, we need empirical research to also consider questions/tasks of this early phase. Currently, this seems to be underrepresented. Thus, we argue that research about the effects of comprehension technologies should put more emphasis on studying and supporting early phase comprehension.

Consider long-term orientation. Similarly, many empirical studies on comprehension, such as the work of Karahasanović and Thomas [14], focus on tasks that can be performed within a couple of hours. While these works are important and lead to relevant insights, they show partially different results than our findings. For example, Karahasanović and Thomas find difficulties with program logic and inheritance, while our students struggled with aspects of scale like used frameworks. Also, in contrast to Wettel et al. [31] our students are not convinced of the benefit of code cities. Therefore, we argue that research also needs to find a way to study comprehension activities that happen with a long-term goal in mind. The above comparison indicates that developers preparing to work with a system over a longer time span (need to) approach comprehension differently than developers who will only have a limited interaction with a system. We would like to see research evaluating and comparing comprehension techniques in such long-term oriented set-ups.

Study the right timing for a technology. Interestingly, our data indicates that some technologies become more useful as the users reach some understanding of the system (see Section 5.12). This suggests that there might be more or less optimal points in time to use different technologies. Furthermore, we find for the same technologies teams that consider that they used them too early and teams that considered them useful in this early stage when used in combination with other techniques, such as code reading. Thus, we advocate for research on prerequisites for the use of comprehension technologies.

Study the effects on intrusive comprehension approaches. A small number of our students chose an intrusive approach to comprehension by changing or refactoring a system to improve their understanding. This is a strategy that can also be observed elsewhere. For example, the Linux Eudyptula Challenge[16] included a set of tasks to change code related to the Linux kernel to slowly familiarize new developers with the code-base. However, we yet need to understand why such intrusive comprehension approaches are helpful. There is a need to develop strategies for novices to use code changes and refactoring to gain an understanding of a system for which no familiarization tasks were defined by senior developers.

6.3 Threats to Validity

Below, internal and external threats to validity are discussed.

Internal threats to validity. The first threat to interval validity is the question whether we are studying the right population, i.e. whether students can have reached a sufficient understanding of the system after five weeks. We mitigate the concern by placing the focus group discussions after the first group milestone, which includes an assessment targeted at ensuring that teams started their comprehension work and reached a first understanding of the systems structure. In addition, the results have to be seen as a study of an early comprehension phase, only.

A second threat is the amount of lecture time spent on the different comprehension technologies. There is the inherent threat that students rate technologies worse which they did not understand, fully. In general, it would be good to spend more time on each technology. However, we use active lectures and an assignments to make sure that students at least have used the technologies and understand the basics of them. Furthermore, we reduced the number of tools taught over the last four years to try and match the students' capacity for learning. As our results indicate, different techniques do not exist in isolation. Thus, participants' impressions of particular techniques might depend on other techniques taught/used at the same time. Future studies need to further investigate the effect of teaching/using different technologies simultaneously.

The two systems used in the two years were different in complexity and used technologies. It is possible that the system choice affects the answers given. For example, the topic that understanding technologies and frameworks is considered important came up during discussions in both years. However, it is thinkable that it would not have played a role, if we had used simpler systems.

Another threat to this research is that students (just like most people) might go for the simplest solution first. We tried to minimize the effect with the individual milestones, triggering them to try out even more difficult techniques. Nonetheless, our results should not be used for quantification of preferences.

Finally, students might fear to say something wrong during the reflection sessions. This threat is difficult to completely avoid. However, mitigated the risk by doing the grading deliberately before that reflection session and not after (the points were given directly by the TAs in the system, visibly for the students). Note that as this checkpoint is normally passed by all teams, reducing the risk that answers were caused by student's frustration. Furthermore, we explained that the reflections aim is to learn and that we expect that some techniques might not work for all teams.

External threats to validity. One should be careful when drawing conclusions on the state of tooling, such as reverse engineering tooling from this data. The reason is that the students mostly did not use commercial state-of-the art tools, due to the educational set-up. Therefore, our results can only be seen as an indication about the concepts behind different technologies.

Finally, it would have been great to cover more comprehension techniques, e.g. regarding visualization and simulation of software, as done by Čisa et al. [5, 6]. Our results are not necessarily generalizable to all types of software visualization, documentation generation, e.g. software summarization, or reverse engineering technologies. Nonetheless, we think that our findings revealed some interesting insights, which can be a good basis for future research.

[16]Linux Eudyptula Challenge http://eudyptula-challenge.org/

7 CONCLUSION

In this paper we presented the design of part of a course on software evolution with focus on software comprehension. We studied students' experience with different software comprehension techniques and their judgment of these techniques in a qualitative manner. Our results indicate that technologies are used in combination, e.g. to guide code reading efforts, and that the timing of a technology's use might impact its perceived benefit. We suggest that future work should aim at teaching comprehension explicitly as a set of different steps that together form a process and to create a real comprehension need.

Acknowledgments

We are grateful to all participants of our study!

REFERENCES

[1] Rawan AbuLail and Mohammad Shkoukani. 2013. The Reality of Software Reverse Engineering Education in the Jordanian Universities and How to improve it. *International Journal on Computer Science and Engineering* 5, 3 (2013), 174.

[2] MA Austin and MH Samadzadeh. 2005. Software comprehension/maintenance: An introductory course. In *18th International Conference on Systems Engineering (ICSEng'05)*. IEEE, 414–419.

[3] Andréa Sabedra Bordin and Fabiane Barreto Vavassori Benitti. 2018. Software maintenance: what do we teach and what does the industry practice?. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering*. 270–279.

[4] Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. 2010. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2503–2512.

[5] Sanja Maravić Čisar, Robert Pinter, and Dragica Radosav. 2011. Effectiveness of program visualization in learning java: a case study with jeliot 3. *International Journal of Computers Communications & Control* 6, 4 (2011), 668–680.

[6] S Maravić Čisar, R Pinter, D Radosav, and P Čisar. 2010. Software visualization: The educational tool to enhance student learning. In *The 33rd International Convention MIPRO*. IEEE, 990–994.

[7] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. 2010. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering* 37, 3 (2010), 341–355.

[8] Massimiliano Di Penta, RE Kurt Stirewalt, and Eileen Kraemer. 2007. Designing your next empirical study on program comprehension. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*. IEEE, 281–285.

[9] Mohammad El-Ramly. 2006. Experience in teaching a software reengineering course. In *Proceedings of the 28th international conference on Software engineering*.

[10] Gregor Gabrysiak, Markus Guentert, Regina Hebig, and Holger Giese. 2012. Teaching requirements engineering with authentic stakeholders: Towards a scalable course setting. In *2012 First International Workshop on Software Engineering Education Based on Real-World Experiences (EduRex)*. IEEE, 1–4.

[11] Gregor Gabrysiak, Regina Hebig, Lukas Pirl, and Holger Giese. 2013. Cooperating with a non-governmental organization to teach gathering and implementation of requirements. In *2013 26th International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 11–20.

[12] Rodi Jolak, Khan-Duy Le, Kaan Burak Sener, and Michel RV Chaudron. 2018. OctoBubbles: A Multi-view interactive environment for concurrent visualization and synchronization of UML models and code. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[13] David H Jonassen and Lucia Rohrer-Murphy. 1999. Activity theory as a framework for designing constructivist learning environments. *Educational technology research and development* 47, 1 (1999), 61–79.

[14] Amela Karahasanović and Richard C Thomas. 2007. Difficulties experienced by students in maintaining object-oriented systems: an empirical study. In *Proceedings of the ninth Australasian conference on Computing education-Volume 66*. Australian Computer Society, Inc., 81–87.

[15] Jussi Koskinen. 2009. Seminars on Software Maintenance and Evolution: An Empirical Study of the Background Factors Affecting Student Success. *The Open Software Engineering Journal* 3, 1 (2009).

[16] Jussi Koskinen. 2010. Experiences from software maintenance seminars: Organizing three seminars with 127 groups. In *Technological developments in education and automation*. Springer, 339–344.

[17] Debora Maria Coelho Nascimento, CVFG Chavez, R Bittencourt, Sao Cristovao-SE-Brazil, and Feira de Santana-BA-Brazil. 2013. Learning Software Evolution: Curriculum, Approaches and an Experience Report. *Fórum de Educação em Engenharia de Software (FEES)* (2013).

[18] Martin J Packer and Jessie Goicoechea. 2000. Sociocultural and constructivist theories of learning: Ontology, not just epistemology. *Educational psychologist* 35, 4 (2000), 227–241.

[19] Ricardo Pérez-Castillo, Ignacio García-Rodríguez de Guzmán, Felix García, and Mario Piattini. 2013. A teaching experience on software reengineering. In *2013 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 1284–1293.

[20] Maksym Petrenko, Denys Poshyvanyk, Václav Rajlich, and Joseph Buchta. 2007. Teaching software evolution in open source. *Computer* 40, 11 (2007), 25–31.

[21] Keith R Pierce. 1997. Teaching software engineering principles using maintenance-based projects. In *Proceedings Tenth Conference on Software Engineering Education and Training*. IEEE, 53–60.

[22] Margot Postema, Jan Miller, and Martin Dick. 2001. Including practical software evolution in software engineering education. In *Proceedings 14th Conference on Software Engineering Education and Training.'In search of a software engineering profession'(Cat. No. PR01059)*. IEEE, 127–135.

[23] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. 2017. Comprehending studies on program comprehension. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 308–311.

[24] Georg Siemens. 2005. Connectivism: A learning theory for the digital age. International Journal of Instructional Technology and Distance Learning. (2005).

[25] Jonathan Sillito, Gail C Murphy, and Kris De Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 23–34.

[26] Shahida Sulaiman, NurAini Abdul Rashid, Rosni Abdullah, and Sarina Sulaiman. 2008. Supporting system development by novice software engineers using a tutor-based software visualization (TubVis) approach. In *2008 International Symposium on Information Technology*, Vol. 4. IEEE, 1–8.

[27] Mikael Svahnberg, Aybüke Aurum, and Claes Wohlin. 2008. Using students as subjects-an empirical evaluation. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. 288–290.

[28] Claudia Szabo. 2014. Student projects are not throwaways: Teaching practical software maintenance in a software engineering course. In *Proceedings of the 45th ACM technical symposium on Computer science education*. 55–60.

[29] Guy Tremblay, Bruno Malenfant, Aziz Salah, and Pablo Zentilli. 2007. Introducing students to professional software construction: a" software construction and maintenance" course and its maintenance corpus. *ACM SIGCSE Bulletin* 39, 3 (2007), 176–180.

[30] Arie Van Deursen, J-M Favre, Rainer Koschke, and Juergen Rilling. 2003. Experiences in teaching software evolution and program comprehension. In *11th IEEE International Workshop on Program Comprehension, 2003.* IEEE, 283–284.

[31] Richard Wettel, Michele Lanza, and Romain Robbes. 2011. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*. 551–560.

[32] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.