Reading to Write Code: An Experience Report of a Reverse Engineering and Modeling Course

Brooke Kelsey Ryan, Adriana Meza Soria, Kaj Dreef and André van der Hoek
{brooke.ryan,amezasor,kdreef,andre}@uci.edu
Department of Informatics
University of California, Irvine
Irvine, CA, U.S.A

ABSTRACT

A substantial portion of any software engineer's job is reading code. Despite the criticality of this skill in a budding software engineer, reading code—and more specifically, techniques on how to read code when integrating oneself into a large existing software project—is often neglected in the typical software engineering education. As part of a new professional Master of Software Engineering at [anonymous institution], we designed and delivered a "reading to write code" course from the ground up. Titled Reverse Engineering and Modeling, the course introduces students to techniques they can use to become familiar with a large code base, so they are able to make meaningful contributions. In this paper, we briefly introduce the Master program and its underlying philosophy, articulate the course's learning outcomes, present the design of the course, and provide a detailed reflection on our experiences in terms of what went well, what did not go well, what we do not know yet, and what our next steps are in preparing for the forthcoming incarnation of the course in Spring 2022. In so doing, we hope to provide a baseline together with lessons learned for others who may be interested in instituting a similar course at their institution.

KEYWORDS

Reading code, software understanding, large open source systems

ACM Reference Format:
Brooke Kelsey Ryan, Adriana Meza Soria, Kaj Dreef and André van der Hoek. 2022. Reading to Write Code: An Experience Report of a Reverse Engineering and Modeling Course. In *44nd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510456.3514164

1 INTRODUCTION

In his essay "Reading to Write", author Stephen King famously advises aspiring writers:

> *If you want to be a writer, you must do two things above all others: Read a lot and write a lot. There's no way around these two things that I'm aware of, no short-cut. [22]*

The same could be said for writing software. Indeed, the point is frequently made by professional programmers on all sorts of fora (e.g., [37], [32], [6]).

In this context, it is interesting to observe that existing curricula in software engineering, or computer science more broadly, remain largely silent on the topic. Introductory courses may include exercises in reading code so to become familiar with language syntax and program structure, and capstone courses may implicitly expect students to read code when they work on a large-scale legacy or open source system, yet reading code as a fundamental topic of explicit and sustained focus throughout a course appears elusive.

In the context of a new professional Master program in Software Engineering at [anonymous institution], the need for a full-fledged "Reading to Write Code" course was identified as a necessary part of the curriculum. Initially, we thought that development of such a course could take inspiration from similar courses elsewhere, as we thought that, surely, such courses exist. Despite our best efforts searching, however, both with Google and through our personal network of contacts, we did not find any. As a result, we designed and delivered the course from the ground up, titling it Reverse Engineering and Modeling for reasons to be discussed later.

In so doing, we had to decide upon the overarching philosophy of the course. Stephen King expanded on his famous quote as follows:

> *So we read to experience the mediocre and the outright rotten; such experience helps us to recognize those things when they begin to creep into our own work, and to steer clear of them. We also read in order to measure ourselves against the good and the great, to get a sense of all that can be done. And we read in order to experience different styles.*

Yet developing an equivalent such sense for coding is not as straightforward as reading a novel, nor as convenient as opening up a book whilst on the treadmill (as King later recommends). One approach to the challenge of course design may be from a fine arts perspective, as advocated by Gabriel [15], wherein students examine existing code, discussing its merits and shortcomings, and build up a repertoire of coding craftsmanship that are excellent, very nice, good enough, mediocre, and bad. We could also think about it from the perspective of learning how to code, as in a reading-focused introduction preceding any programming. Neither perspective, however, aligns well with the nature of the overall program. The former necessitates smaller classes and stops short of giving the students concrete experiences that can be put on their resume, which, as a professional Master program, is a must. The latter does not align well with the incoming cohorts: in being as a graduate program,

ICSE-SEET '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9225-9/22/05.
https://doi.org/10.1145/3510456.3514164

the admission requirement is for them to have programming experience, whether through a formal education or informally acquired (e.g., self taught, bootcamp).

We, thus, settled on a third approach: the pragmatic, professionally oriented objective of *learning how to read the code of an existing, large-scale system to become an effective contributing member of its community*. While all students entering the Master have prior programming experience, few have experience with what they are likely to encounter when they graduate: an existing, large-scale system that has been worked on for years by numerous developers who have established practices of how to work together. The course, then, sought to give them such an experience in a single quarter (10 weeks), as anchored by theory and practice in reading code.

This paper presents our experiences in designing and delivering the course. It is based on the second offering of the course, as the first led to a number of improvements that we feel have created a stable base from which to move forward.[1] The paper is intentionally structured as an experience report: our aim is to share the design of the course and discuss what went well and what did not go well.

The remainder of this paper is organized as follows. Section 2 briefly reviews relevant background material. Section 3 introduces the Master of Software Engineering program and its overarching positioning as a degree program. Section 4 presents the learning outcomes that we set in creating our course, while Section 5 articulates how we sought to address those learning outcomes in the pedagogical design of the course. Section 6 reflects on our experience in delivering the course with a series of lessons learned. Section 7 concludes the paper with an outlook at our future work.

2 BACKGROUND

A substantial portion of any software engineer's job is reading code [27]. In a typical day, they may read code to understand where to fix a bug or add a feature, how someone else has implemented some functionality, or to resolve a merge conflict. They may also need to read code outside of the project to which they are contributing, in order to understand how to use a particular library or to investigate which of several libraries suits their goals better.

In Begel and Simon's 2008 study of college graduate software engineers at Microsoft[5], newly hired developers were observed to have significant difficulty in the technical and social nuances of integrating with existing large systems, a skill that participants acknowledged was not taught to them at university. More recent studies on newcomers to large software systems confirm and expand these findings, observing that individuals not only struggle with the code, but also encounter significant challenges in terms of the social context in which they are operating (e.g., [39], [38], [4]).

How exactly software developers read code in practice, then, has become of significant interest to the software engineering research community. While a comprehensive review is out of scope, Sillito et al. [35] studied the kinds of questions developers ask "of the code"; that is, what are the kinds of things that they want to know? As another example, Roehm et al. [33] performed an observational study of 28 developers to identify the strategies they follow in building their understanding of the code. LaToza et al. [23] examined how developers build and maintain their mental models of the code, with a key observation that they often rely on asking one another to deal with missing knowledge. More recently, too, several studies have used fMRI techniques to study the cognitive processes of programmers at work (e.g., [18], [14], [21]).

Information foraging theory, perhaps the most well-known distillation of the aforementioned observations, essentially likens the search for information in the source code to an animal's hunt for its prey – both try to optimize the energy expended versus the expected payoff [12], [28].

Much work has gone into the design of novel tools to facilitate program comprehension. Examples include techniques that summarize methods to make them easier to digest (e.g., [13]), advanced visualizations (e.g., Code City [41]), tools that help maintain mental models [23], and tools supporting specific tasks (e.g., Tarantula [19]). While these are important in their own right, we consider them to be too specific and advanced for purposes of our class. Therefore, we introduce tools such as Call Graph and sequenceDiagram that are more standard, opting to then point the students to more advanced tools in the extra materials to explore independently.

For other topics in the course (e.g., patterns, social context, modeling), the story is much the same. A great amount of research exists around them; each, in many ways, is represented by an entire subfield of software engineering research. We again chose to structure the course as introducing the foundations upon which the students later, on their own during the course or after they have entered industry, can explore the more advanced and nuanced body of knowledge that may assist them in their jobs or provide more insight into what they do on a day to day basis.

While constructing our course, we surveyed the literature on related course work, and found that courses focus on either: (1) code reading as a necessary component, or (2) experience with open-source systems. Several studies on programming coursework have indicated the importance of providing a curriculum on code reading comprehension ([6], [7]). Hillburn et al. [17] in particular advocate for active learning pedagogical methods in code reading, which is precisely what we offered at the scale of a professional Master program and as applied to large scale open source systems. In regards to leveraging open source projects, even prior to the inception of GitHub in 2008, courses sought to use active, publicly available software systems as an educational tool [3]. The use of open-source for courses like capstones has become very popular in software engineering curricula (e.g., [29], [24], [10]). Smith et al. [36] cite the difficulties inherent in screening sufficiently complex and active open-source systems suitable for student projects, a sentiment we also found to be true.

The Software Architecture course at TU Delft [40], was most influential to the development of our course and shares a number of similarities in its approach. Both courses include the use of team-based open-source software projects, an emphasis on a balance of technical and social skills, and requiring students to contribute to their project via pull requests. However, the TU Delft course remains anchored in software architecture, culminating in an online booklet describing architectural principles of the chosen project. Our course does touch upon software architecture, but strongly

[1] Note that the first incarnation was in person while the second was, due to COVID-19, online. The online course, however, was still delivered live, with in class exercises, in class reflection on the exercises, guest speakers, and more, much as it would have been had it been in person. As such, we do not believe that the remainder of what we discuss is particularly colored by the experience of teaching online.

focuses on the theme of "reading to write" and giving students practical ways of reading and understanding a large code base.

3 CONTEXT

Our Reverse Engineering and Modeling course is part of the new professional Master of Software Engineering degree program that was launched Fall 2019 with the arrival of the first cohort of students. The program is an "advanced 15-month professional degree program designed to train software engineers coming from a variety of backgrounds" [1]. The program's course sequence and pedagogy were constructed to not only welcome those who already have obtained an undergraduate degree in some form of computing, but also those who had a different education and acquired some initial programming skills and are interested in switching careers to software engineering. The program accomplishes these goals through an intensive set of programming-oriented mini courses in, among others, applied data structures and programming, database programming, GUI programming, and applied data analytics. Students are advised and coached on the mini-courses they are taking. Advanced students benefit by polishing their skills and being able to dive deeper into more advanced topics; novice students benefit from quickly getting up to speed in the basics, so they catch up sufficiently to the more advanced students in the core topics. The result has been a cohort of 35 students in year 1, with 22 students (63%) career changers, a cohort of 46 students in year 2, with 16 students (35%) career changes, and a cohort of 57 students in year 3, with 35 students (61%) career changers. All but one student from the first cohort successfully graduated (with the other finishing up part-time this quarter) and all but three students from the second cohort are on track to graduate according to schedule this quarter.

Table 1 provides an overview of the full curriculum, which also mandates a supervised internship in the summer between Quarters 3 and 4. Each of the courses in quarters two through four is required; all students take these courses in lockstep with one another. The Reverse Engineering and Modeling course is offered in Quarter 3, intentionally later in the curriculum so that it can build on prior experiences. It particularly relies on material in the Distributed Software Architectures course, because many of the open source systems encountered in the Reverse Engineering and Modeling course are distributed in nature. The course, as we shall discuss later (see Section 5.3, week 8), also builds on the prior Software Testing and Debugging course by using test cases as a tool to understand code.

The course is strategically offered before the summer internship and Capstone Project, as students are likely to encounter existing systems to which they will be expected to contribute in both. In the summer, it will be at whichever company that hosts their internship (e.g., Oracle, Kaiser Permanente, Facebook, Uber, and SAP) and in the Capstone Project, it will be through whichever outside entity acts as their project sponsor (examples have been Expedia, Apple, Blizzard Entertainment, Sony, and Oppo). In either case, the systems they work with tend to be large and they have to become familiar quickly to make meaningful contributions throughout the short period of 10 weeks in either. This is what the Reverse Engineering and Modeling course prepares them for.

The student body in the Master program is relatively diverse. In addition to having a large percentage of career changers (see above), 33%, 41%, and 43% were female across the three cohorts, and 5% consistently year after year under-represented minorities. The latter is a percentage that we are actively seeking to increase.

Finally, a note on why the course is titled Reverse Engineering and Modeling instead of Reading to Write Code, or some variant thereof. The reason is primarily historic, as the creation of new degree programs is a multi-year effort at our institution with many steps of discussion and approval. The original concept of the course, when the proposal to the campus for a new degree program was conceived, had elements of understanding code in it (therefore the Reverse Engineering part of the title) as well as a fairly strong doses of design work (hence the Modeling part). When the actual classes were to be developed, it became clear that some but not all of the design content could usefully be moved to other courses, meaning that this course could more strongly focus on the reading and understanding code aspects.

4 LEARNING OUTCOMES

The overarching goal of the course is to educate students in specific techniques of reading the code of an existing, large-scale system so to become an effective contributing member of its community. Underlying this goal is our desire to thereby lower the psychological barrier involved in having to approach an unfamiliar, large-scale code base. Many of the students in the program—especially those who are switching careers and do not have a formal education in computing—have never encountered a large-scale system before. To them, the thought of having to actually make sense of something that is so large that they cannot feasibly print or read it all, let alone to be asked to make changes, is daunting.

Towards this goal and desire, then, we identified a set of eight concrete learning outcomes that governed the design of the course.

Learning outcome 1. Be able to find where some functionality is implemented. A core task that any software developer faces is to figure out where in the code base some feature is implemented, so they can augment that feature or fix a bug in its behavior. In a large code base, this can be quite challenging and take significant time and effort. A core reading skill to develop, then, is to be able to locate the lines of code or, at a larger scale, those parts of the code where certain functionality is realized.

Learning outcome 2. Be able to apply different strategies to understanding code. Watching professionals at work making sense of a code base can feel daunting and at the same time haphazard as they move through the code base and make inferences at what appears lightning speed. Underneath their approaches, however, are common strategies for how they choose to navigate and what they choose to look at. We want our students to understand that such strategies exist and apply different ones when so applicable to the task at hand.

Learning outcome 3. Be able to leverage information that comes from many sources. The code base itself is not the only source of relevant information when it comes to understanding it. Comments, design documents, issues in the issue tracker, test cases, API documentation, and indeed people are among the many potential sources that one can and should rely upon when approaching a

Table 1: Master of Software Engineering Curriculum.

Quarter 1	Quarter 2	Quarter 3	Quarter 4
Programming courses (six total)	Programming Styles Distributed Software Architectures Software Testing and Debugging	User Experience and Interaction **Reverse Engineering and Modeling** Software Security and Dependability	Career and Entrepreneurship Project Management Capstone Project

new code base (and often also when approaching a new part of a code base with which one is partially familiar).

Learning outcome 4. Be able to work with the reality that external information is not in sync with the latest version of the source code. While external information can be extremely valuable (conform the previous outcome), it is crucial for students to understand that it cannot be taken as an absolute truth, for the simple fact that it often becomes out-of-date over time. That said, there still can be significant partial value, but one should always be skeptical and check with the source code as to whether what is being documented is still accurate in the latest version of the code.

Learning outcome 5. Be able to contribute while adhering to the unique social context of a software system. In order to make actual contributions to a software system, it is important to adhere to the broader social context in which it is being developed. Just submitting a pull request is insufficient; one should do so according to the coding, documentation, and process standards that are expected – whether those standards are explicitly documented or are part of an informal culture that has emerged and has to be learned informally. Moreover, when interpersonal interaction is needed (as it is often when the initial pull request is returned with requests for improvement), it is similarly important to be able to follow the social conventions of the project.

Learning outcome 6: Be able to assess the value of existing code. A tendency especially among newcomers is to think that one can do better than the current code and, thus, to want to rewrite or to talk unfavorably about the code. This misses the point, as existing code carries with it a lot of history and implicit knowledge that must be honored; it is shaped as it is through careful deliberation by those developers who came before. We want our students to understand this and learn how to make changes only when absolutely needed.

Learning outcome 7: Be able to make meaningful contributions. While reading code for one's own learning is an important practice, ultimately, the expectation of the future employers of our students will be that they will actually engage in furthering the software on which they are working. As part of our course, then, we want them to practice making actual changes to a large-scale software system that is not their own.

Learning outcome 8: Be able to use tools where and when so needed. Reading code and making changes to it is a cognitive intensive and often largely manual task. That said, many different tools exist that can ease the task, at least to a degree. We want our students to develop a basic understanding of what kinds of tools exist and what kind of value these can bring.

Not a learning outcome, but a major secondary goal in the course design was to structure projects such that they become resume items that can carry weight with potential recruiters and interviewers. We sought to provide valuable and relevant experience in that regard by designing the course such that students become familiar with a real, large scale software system, study it from a variety of angles (including code, architecture, patterns, issues, and test cases, as we will see below in Section 5.3), and actually make and submit pull requests that are taken under consideration by the project developers. We believe that demonstrating in interviews that they learned and applied a principled approach to an unfamiliar system, integrated with the project social context, and ultimately contributed a piece of code to the system, gives them an important advantage in securing a future position.

Finally, it is important to note what this course is not. It was expressly not our goal to integrate the latest and greatest research results and especially not the many research tools that exist for reverse engineering, program comprehension, software visualization, and more. While these research tools are important, we felt that for purposes of this course we should stick with the more basic tools that we know the students could bring into and use in any organization. The course is also not a perfect recipe. While various strategies are offered, many heuristics and techniques are discussed, ultimately, the success one has in reading code depends as much on the system under study as one's own personal skills. We want the students, thus, complete the course with a realistic understanding of both the possibilities and the challenges involved.

5 COURSE DESIGN

With these learning objectives in mind, we settled on a course design in which students assemble into small teams, with each team adopting a single, large scale open source project that becomes their practice ground for the duration of the class. With each week of new material, we move from reading and understanding increasingly complex aspects of the system to eventually making code and other contributions, as further detailed in Section 5.3.

In some ways, the guided exploration that results represents a typical onboarding experience [20]. In other ways, our approach deviates. First, it spends more than a few weeks reading only, whereas in the typical onboarding experience the focus is on making small contributions as early as possible. Second, in onboarding, new employees are typically paired with more senior colleagues who guide them; such colleagues are absent here. Finally, while in onboarding, a new employee is often familiarized with a small part of the system and then spends considerable time there honing their skills, in our course we intentionally engage the students with different parts of their systems and at different levels of abstraction.

In the sections below, we first briefly introduce the overall pedagogy we applied throughout the course. Then we detail the basic structure to which each lecture adhered, before we introduce and discuss the full set of topics week-by-week.

5.1 Overall Pedagogy

We adopted an active learning [8] approach, with some lecturing of the underlying basic theory and time allotted to practice the theory on a small example code base during lecture. Given that lectures were once a week for three hours, this not only enabled students to connect with the material in a hands-on manner, but also allowed for the three-hour block to not be a single, monotonous monologue of new material.

In addition, we adopted a philosophy of not teaching the students everything they need to know. Instead, we focused on introducing the concepts, with the students being responsible for digging deeper as needed. When discussing software patterns, for instance, we introduced what patterns are and what they are good for with the help of two example patterns. We then pointed students to resources where they could read more about patterns and study additional patterns on their own. The assignments from week to week would lean not just on what was discussed in lecture, but also forces them to engage with the additional material. Again in the case of software patterns, the assignment was to identify five different patterns in the software being studied, requiring the students to go beyond the two that were discussed in lecture and self-learn about others.

We also considered teamwork essential, given that the vast majority of graduates end up in positions where they function as part of a larger team. While we could have randomly assigned the students to teams, we decided to let them choose teams based on mutual interest in terms of the domain of the software they study. We did so for two reasons. First, by allowing students to choose a software system in a domain of interest (e.g., Bitcoin related, database related, genomics related), we give them a more meaningful experience with respect to future interviews and job opportunities. Second, by assembling teams based on interest rather than on friends, we still achieve a reasonable amount of working with new people. All teams consisted of four or five students.

Finally, we wanted to ensure that the course content reflected the real world. As the course is part of a professional program, this should not come as a surprise. We therefore engaged in two critical steps. First, we surveyed a large number of alumni from our other programs before designing the course, asking what they considered relevant and important content, how they personally went about navigating an unknown code base, what tips and tricks they might have, and what skills they think are necessary for students to possess at completion of the course. We used the collective responses to determine the topics being taught. Second, we brought in weekly guest speakers, who complemented the lectures with professional perspectives (e.g., a Google employee walked us through an actual code review on real Google code; a ZocDoc employee showed bad code they had written and what consequences it had; an alumnus who founded a successful start-up talked about how students could parlay their experiences in the course into strategies for interviewing).

5.2 Basic Structure of Each Lecture

While the topic of the course varied each week (see Table 2, as discussed in detail in the next subsection), we structured the lectures to follow a common pattern that we briefly introduce and motivate here. Table 3 summarizes the common pattern.

First, we started each lecture with a few quotes from professionals in the real world, as collected in our conversations with our alumni. Each week, we selected the quotes to match the topic of the ensuing lecture. For instance, one of the quotes preceding the lecture on mental models is from a Research Staff Member at the MIT-IBM Watson AI Lab:

> *Often I use a debugger to understand information flow and a diagramming tool to help me build a mental model of the system.*

As another example, the following quote is from a Senior Software Engineer at ZocDoc, a quote that we used to open the lecture on leveraging test cases as a way of understanding source code:

> *When you are ready to dig into more low-level details, check existing unit/integration tests. Run them to verify your own understanding of each part of the system. If there are no tests, try to write them yourself, you will do a public good and it will increase your understanding of how differently pieces work independently and together. By all means, make sure there is enough test coverage before you modify any of the code!*

These quotes served as the basis for a brief class discussion based on the instructor asking the students to interpret the quote and explain what they think it may mean. This helped both in establishing the validity of the topic as relevant to their future profession and in setting the expected level of engagement for the subsequent lecture. The overall set of quotes at our disposal proved remarkably relevant to the course, with a multitude to choose from for each topic.

Following the quotes, we briefly recapped the topic of the prior lecture. After, we revisited the project assignment from the prior week to debrief on the experience: what went well for the teams, what represented hurdles, and what did they learn? It was in this debrief that a significant amount of learning across projects took place. Students heard the experiences of other teams, for instance when one team shared of their open source project being ran by a particularly voracious maintainer who would close issues lightning fast (leaving none for the team) or when another team explained why they found it difficult to precisely articulate the boundaries of where a feature was and was not implemented in a system. These experiences sometimes led to sympathy being expressed (as in the former case), but nearly always resulted in discussion that expressed recognition and validation – other teams would experience similar issues in their projects.

We then switched to the topic of the week, usually introducing the underlying theory for about twenty to thirty minutes. For the lecture on basic strategies of code navigation, for instance, we introduced the students to both typical strategies (top-down comprehension, bottom-up comprehension, systematic comprehension, opportunistic comprehension, per [9]) as well as information foraging theory [12, 28] to explain why different situations may call for a different approach.

Immediately after introducing the theory, the lecture switched into short practical exercises where applicable. Students performed these exercises in randomly-assigned teams, with their output being captured through Jamboard [31] so that the instructor could bring up any of the random teams' results to discuss in class. The primary open-source system we used for these exercises was JPacMan, a

Table 2: Course Design.

Week	Interactive lecture	In-class Practice	Team Project Work
1	Course goals Course logistics Expectations for engagement	Baseline questions and brainstorming	Setup environment Build sample systems Form team
2	Basic comprehension strategies Information foraging theory Hypothesis-driven vs. inference-driven	JPacMan (fix) JPacMan (change) JPacMan (learn)	Choose a system >100KLOC
3	Mental models Mental simulation	JPacMan (locate a feature) JPacMan (understand a feature)	Locate a feature (alone) Locate a feature (together)
4	Externalizing mental models Structural versus behavioral modeling UML		Class diagram of entire system Locate two essential features Document those features
5	Social context Code change principles	Brainstorm how you would discover Imagine you are a core developer	Choose three open issues (alone) Choose three open issues (together) Answer social context questions
6	*Midterm*		
7	Design patterns	JPacMan (factory) JPacMan (observer) JPacMan (should we refactor)	Identify five design patterns Code up your first issue Submit pull request
8	Reading test cases Running test cases Writing test cases	JPacMan (Board test cases) JPacMan (LevelTest)	Write three test cases Submit a pull request for each Start working on two more issues
9	Design cycle Design principles	Traffic (brainstorm interfaces/classes) Traffic (most important/difficult questions) Traffic (actually design it)	Model a new feature
10	Design principles (revisited) History	Traffic (instructor demonstration)	Submit final two pull requests
11	*Final (optional)*		

Table 3: Structure of a Typical Lecture.

Topic	Duration	Purpose
Quotes from professionals	5 minutes	Orient students to the real world concerns of this lecture
Recap theory from last week	5 minutes	Ensure prior materials are understood
Reflection on last week's project assignment	15 minutes	Share insights across teams to learn from each other's experiences
Theory	30 minutes	Provide foundational knowledge on this week's topic
In-class exercises	50 minutes	Practice with a small-scale system and discuss experiences
Tools (as applicable)	10 minutes	Introduce automated support that eases the task at hand
Readings and resources	5 minutes	Offer pointers to further background reading and free online resources
Expert practices	10 minutes	Provide vignettes capturing typical behaviors of experts
Project assignment	5 minutes	State and clarify expectations for the team project for the upcoming week
Guest speaker	45 minutes	Learn from a professional, their work, and insights

small Java project that was designed and built at TU Delft for educational purposes [42]. For the lecture on test cases, for instance, the first exercise was for the students to write down what they learned and what open questions remained after reading the test cases for a particular subpart of the system and juxtaposing those test cases with the code. To the students, this was an eye-opening experience as they already had been working with JPacMan for weeks and thought they understood it well; using the test cases put some aspects of the system and how it worked in a new light, particularly illuminating to the students was that their understanding thus far had been at a higher level that obfuscated important details. As another example, Figure 1 presents two Jamboards resulting from the exercise in lecture 3 where we asked the random student teams to draw a (mental) model of where scoring is implemented in JPacMan. Note the two very different approaches, which led to a class discussion of what the different teams were trying to capture and why. Note also the embellishments in the second Jamboard, which we take as a strong sign of student engagement. Indeed, these kinds of embellishments became part of the class culture when using Jamboard with a friendly competition as to who could get theirs acknowledged by the instructor.

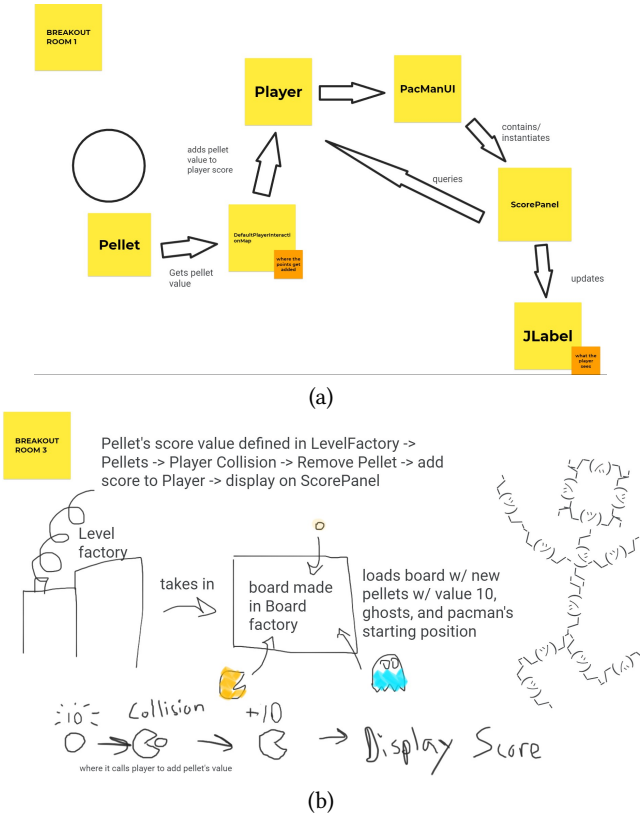

Figure 1: Two Jamboards Produced in Lecture 3.

Crosscutting the theory and in-class practical exercises, we introduced useful tools when so appropriate. We required the students to use IntelliJ, which they were already familiar with but did not know how to use to the fullest extent. We introduced the students on how to follow call sequences using the native features of IntelliJ and also introduced them to a variety of plug-ins, including Git, Statistic, Call Graph, and sequenceDiagram. We also showed them how to use the UML features of IntelliJ Ultimate, which was freely available to them by virtue of being a student.

Rounding out the lecture component each week, we provided pointers to additional reading and the introduction of a few expert practices drawn from the Software Design Decoded book [30], as adapted to code reading. These exercises mirror observed behaviors of expert software designers (e.g., focus on the essence, dig as deep as needed, externalize their thoughts) and serve as a benchmark for students to ask themselves and their team whether they are engaging in their project in ways that professionals would expect them to engage.

We then gave students their homework assignment for the week, according to the schedule shown in Table 2. An important component of the homework was that the students needed to maintain a personal diary, logging when they worked on anything related to the course, who they worked with on that activity (so we could verify team participation if issues started to arise in a team), and what new insights they gained from engaging in the activity. This to expressly insert a reflective step in their learning process.

Finally, we then introduced the guest speaker, who would talk to the students for a planned 45 minutes, but often an hour or longer as some subset of students would stick around past the time allotment for the course to ask questions and hear more stories from the trenches. Some of the guest speakers had prepared materials beforehand, others talked off the cuff, and yet others were content with a brief introduction and then answering questions. Among the materials, beyond the two we already mentioned (code review, bad code), a guest pair brought a Lua-based software defined radio software that they reverse engineered with the students and another guest speaker talked of the issue in dealing with large-scale legacy software written in Filemaker Pro.

5.3 Week-by-Week Topics

Table 2 presents the week-by-week structure of the course, particularly highlighting the primary topics of each lecture, the in-class practice associated with those topics, and the team-based project assignments for each upcoming week. Below, we discuss each week.

Week 1: Introduction. In the first week, we welcomed the students and introduced the course by articulating its goals, sharing the logistics as to how it was to unfold, and setting our expectations for engagement in that the class is interactive, requires their participation throughout, and will call upon them and their thoughts continuously. We also explained the tenor of the course, that we expect them to be overwhelmed at first at the sheer scale of the system they will be studying (the actual requirement is 100,000 lines of code or greater), that we expect there to be surprises, and that we understand the work involves a lot of tedious, low-level engagement with the code base. At the same time, we explained that we view the project as a learning experience and that we provide important scaffolding for them to successfully engage. We also asked for their patience, in that what is frustrating and seemingly impenetrable at first becomes much clearer later, with them making actual contributions to the system. By communicating all this up-front and repeating it throughout the course in the context of their progress, we gained the students' trust.

As a first interactive exercise, we went through four Jamboard sessions, each requiring a random team of students to address a prompt. These prompts were, in order: (1) what are some reasons that we need to read and comprehend source code, (2) what is the largest piece of software you have worked on, (3) give examples of when it was easy to read, understand, and perhaps modify a piece of code, and (4) give examples of when it was difficult to do so. This exercise served to set the tone for future lectures and their interactive nature, but also helped benchmark their initial thoughts and prior experiences.

As their homework, each person was to individually install IntelliJ with plug-ins and attempt to download and build a few example systems, so to become familiar with how to do so. The students, too, were instructed to form teams around areas of common interest.

Week 2: Basic strategies. In the second week, we introduced four strategies for reading code: top-down comprehension, bottom-up comprehension, systematic comprehension, and opportunistic comprehension, per [9]). Moreover, we discussed the difference between being at least somewhat familiar with the code or with similar code and not being familiar. In the former case, the process

typically becomes driven by a hypothesis, with the reader searching for beacons that serve as anchors for the hypothesis [34]. In the latter, the process typically becomes inference driven with the reader attempting to chunk smaller bits of understanding into larger behaviors [34]. We then explained the importance of being aware of how one goes about the reading and interpretation process, and that information foraging theory [12] presents a nice conceptual framework for how to think about their actions and how, as they progress with a task, they may well change strategies.

We then introduced—unbeknowst to students—an altered version of JPacMan and asked them to engage in three tasks: (1) fix a bug in how PacMan moves, (2) change the amount earned per pellet, and (3) figure out how JPacMan animates its characters. Each task was performed by a random team, with Jamboard as the canvas. After each task, the instructor would use the Jamboards to initiate impromptu discussions highlighting differences in findings, how the teams went about the respective tasks, and how their approaches fit into the strategies we introduced prior.

As homework, the team had to choose a system of 100,000 lines of code (LOC) or greater that adhered to a number of criteria: the presence of an issue tracker, at least 10 new pull requests over the past month, and be of shared interest to the team. In addition, the system could not be a standalone library, in order to further ensure the repository is being actively contributed to and maintained.

We encouraged the students to pick a project that aligned with their professional software engineering goals and interests. While the vast majority of selected projects were ultimately Java-based (we believe this is because prior courses are primarily taught in Java), the students selected projects that spanned a variety of technologies and interests. Systems of choice included a 3D game engine (JMonkeyEngine), a Bitcoin exchange platform (Bisq), an Android podcast mobile application (AntennaPod), and a citation and reference manager (JabRef). One team expressed a desire to incorporate Python into their professional repertoire, and chose to work on a YouTube download manager implemented in that language (youtube-dl).

The teams' submitted choices needed to be approved by the TA and instructor in order to ensure the system appeared to meet the above criteria. All selected projects did so for size and repository activity, however one team was denied their initial selected choice of project (FastJson) on the grounds that it was too similar to the JSON-Java system that all students had studied extensively in the Programming Styles course in the previous quarter.

Week 3: Mental models and simulation. The third week introduced the topics of mental models and mental simulations – the fact that, while we work to understand some code, we build a picture in our mind of its various pieces as well as how those pieces work together to accomplish some functionality [11, 23]. The mental model is the static part; the mental simulation is the envisioned execution under imagined circumstances. While every developer engages in mental models and simulations, what sets effective developers apart is that they intentionally think about their models, how they construct them, what parts of the code are in (and out, and why), and what kinds of executions they use to essentially "test" their mental models for completeness, accuracy, and more. In this third lecture, then, we emphasize the importance of this unseen activity.

The practical in-class component was designed to make the unseen seen. First, we asked the students to draw a model of where scoring is implemented (static). Then, we asked them how scoring works (dynamic), which caused them to reassess the static model and add important detail. Third, we asked them to co-develop the static and dynamic model by asking how collisions work. Reflection with the instructor after each task focused on the utility of the models drawn: to what degree do they represent the true understanding in one's head and to what degree do they truly illustrate what is happening in the code? Inevitably, during these discussions the students start to realize shortcomings in their understandings.

The assignment for the week built upon the lecture by tasking students with identifying a simple feature in their system and drawing a model of how that feature works. We first asked students to do so alone, and then asked the team to combine its findings. This led to significant clarification and learning among the team members and illustrates the valuable lesson that many minds matter.

Week 4: Externalizing mental models. Building on the topic of mental models and simulation, week 4 introduced the general concept of modeling languages and the specifics of one of those, UML class and sequence diagrams.[2] The lecture focused on introducing UML syntax, with the help of several of the plug-ins that we mentioned in Section 5.2. The primary goal was to instill a sense of code understanding and navigation at a higher level of abstraction than individual lines of code, with UML class and sequence diagrams serving as roadmaps for exploration. We used JPacMan to show that these kinds of visual representations can be a powerful source of insight into the system under study, even without reading the code: what higher level components exist, how are they connected, and what might be flow overall that can be inferred?

As the assignment, the teams had to use the tools to create a UML diagram of their entire system (to finally get a graphical sense of size and scope), mark on the UML diagram where two newly chosen features were implemented, and, under the hypothetical scenario that those features would need to undergo some change to be made by someone else, prepare roadmaps of the features that help someone understand them. This represented a challenge to the teams, as they had to think about what would and what would not make sense to share, and at what level. Most teams eventually understood that what was needed was a higher-level introduction to the feature and how it worked; this, of course, required them to nonetheless understand the code in detail, so to be able to describe the higher level workings.

Week 5: Social context. Week 5 switched gears from actively working with code to understanding its broader context. We introduced social context as "the specific circumstance or general environment that serves as a social framework for individual or interpersonal behavior" [2]. In other words, contributing to any software project requires one to work with other people and, thus, to understand their culture in terms of formal work processes and informal ways of interaction. As the interactive component of this lecture, we asked the students two questions: (1) how would you find out about the state of project / what kind of indicators might

[2]We focused on these two types of diagrams because they can be especially useful in the context of the kinds of changes the students make, which remain fairly small compared to, say, a major architectural refactoring. Moreover, class and sequence diagrams can be automatically produced by tools.

you look at, and (2) imagine you are a core developer on a project, what would you like to see in the pull requests that come your way? It is particularly this second question that led to significant discussion, with the students realizing there is a lot more to it than just producing some code for an issue. The lecture also touched upon key rules of thumb, such as to respect the wisdom embedded in old code, to always verify assumptions when modifying code, to leave the code in a better place than you found it, and so on.

The assignment was for the team to determine the social context by answering a set of twenty-six standard questions. Among others, questions included: what is the number of accepted pull requests over the past month, how many open issues are there today, is there a development forum and if so how responsive are the maintainers, does it appear that the documentation is up-to-date and how do you know, does the project follow any coding conventions, what have been some comments on recent pull requests that were initially rejected, and what are the expectations for submitting test cases along with your code updates. The team also had to select three issues that they could work on next. We again followed a two-stage process: they first had to choose three issues individually and then come together as a team to discuss and select the three (from all identified) they felt would suitably challenge them.

Week 6: Midterm. Week 6 was the midterm, which consisted of questions about the theory presented in the prior weeks and asked them to apply the techniques we had introduced on small snippets of code. We do not further discuss the midterm for space reasons.

Week 7: Design patterns. In week 7 we switched back to code and introduced design patterns, a topic that we knew was of great interest to the students. The lecture explained the history of design patterns (more broadly and in software) and then introduced the original catalogue of patterns by Gamma et al. [16]. Two patterns were explained in detail and, after each explanation, random student teams were to locate those patterns in the JPacMan code. A third exercise asked them to consider whether a part of the code should be refactored into another design pattern.

The assignment was for the students to locate five different and non-trivial (i.e., not Singleton) design patterns in their system. This exercise leverages the fact that the students are already quite familiar with their code base and therefore can focus on learning to recognize patterns through reading the code. This task is relatively easy to perform through some trial-and-error searching, assuming the system uses proper naming conventions (which most open source systems tend to do). Rather than teaching them an exhaustive list of patterns, we thus allow them to gain experience in utilizing and studying external resources (i.e., documentation on patterns other than the ones discussed in lecture), which helps prepare the students for their upcoming summer internships, in which they frequently apply similar skills (i.e., there too they need to consult web pages with API documentation, YouTube videos, Stack Overflow, and other resources to find what they need to know [26]).

They had to also assess whether the patterns were followed faithfully and they had to write up why they thought the developers used each of the patterns (note that for this exercise they could choose design patterns beyond the original 23 from Gamma et al.). Additionally, the team was to code up the first issue and submit a pull request for it. All teams succeeded in submitting such a request. That said, despite the pull requests typically being very small in touching merely a few lines of code, most were returned with comments by the maintainers. As an example, a maintainer of JMonkeyEngine wrote:

> *The comments say that an ArrayIndexOutOfBoundException is thrown, but that's no longer true.*
>
> *Unit numbers start from 0, so the if (unit >= 17) test appears to be off by one. Have you tested this code?*
>
> ..
>
> *To demonstrate how this PR [pull request] would work in practice, I think it should include a test case of a material with 17 textures. Test cases can be added to the jme3-examples subproject.*

These and other comments like it served, in our next lecture, as the anchor for a discussion revisiting social context and expectations. Mistakes made across the board included not commenting sufficiently, changes lacking test cases to demonstrate it worked, wrapping multiple changes in a single pull request, and changes not working for some edge cases – precisely the kinds of topics we had talked about in week 5.

Week 8: Test cases. Although rarely thought of as such, test cases can be a powerful source of understanding a code base, since they specify what should and should not happen and can be traced through the source code to understand where what functionality is implemented. Leveraging the fact that students already were familiar with JUnit from the Software Testing and Debugging course, this lecture reviewed the topic and then asked students to practice reading test cases and the code to which they map to understand aspects of JPacMan (as already mentioned in Section 5.2).

Teams were tasked with the homework of writing three test cases and submitting each as a separate pull request. This allowed them to once again practice interacting with the community. Interestingly, the submission of new test cases was generally appreciated by the maintainers, even though some did send feedback to improve the test cases in one way or another. Teams also had to start working on the final two (adjusted from three as initially planned) issues that they were going to submit as pull requests; these had to be approved by the TA and instructor, so to ensure sufficient complexity.

Week 9: Design cycle and principles. In week 9 we discussed how to navigate making larger changes that require serious upfront consideration of what the best way forward may be. We introduced the design cycle of analyze–synthesize–evaluate as a way of engaging in principled design thinking and exploration of options. As a complementary topic, we talked about design principles, particularly focusing on SOLID [25] as a means to strive for changes that maintain a high level of code quality.

As the practical component, we switched from JPacMan to the students designing a mini traffic simulator on Jamboard. This, as we shall see in Section 6, was not the best pedagogical choice as students ended up being bogged down in understanding a new domain and design prompt. This caused lecture 10 to have to revisit material that we thought we would be able to cover in just lecture 9. We should have asked them to design a change to JPacMan.

As homework, the teams had to model a new feature for their system by documenting the existing state of the software, identifying how and where the impact would be, and discussing why they would go about making the change in the way they proposed.

Week 10: Design principles (revisited) and history. Week 10 was to be a lecture about source code history and how it can be an important asset when it comes to understanding the latest state of the code. We wanted the students to learn how they could identify who had worked on the code before (so in their professional lives they would be able to find out whom to reach out to when so needed), how to read the version history in terms of branches, merges, and pull requests, and how to go back in time to understand why a certain piece of code is shaped like it is. The JPacMan project has exemplary history that could be used to illustrate. Unfortunately, as we discussed, we needed to revisit design principles, which took most of the time of this lecture.

The final assignment was for the team to complete coding and submit pull requests for the final two pull requests. Reception of these pull requests was similarly mixed as for the earlier ones, with a few being accepted and a good number receiving comments from the maintainers. Unfortunately, by the time some of those comments arrived, the quarter had ended and most teams (despite our messaging that these would be important resume items) stopped working on the pull requests if the feedback appeared to involve more work than they were prepared to do of their own free will.

Week 11: (Optional) final. The course had an optional final for those who wished to improve their grades; the final was structured in the same way as the midterm.

6 REFLECTION

As with the first incarnation of the course, we have spent considerable time reflecting on this second offering so to be able to further improve the course when we teach it next in Spring 2022. Below, we first sample the reactions from the students and then offer our thoughts as to what went well, what did not go well, what we do not know yet, and what our plans are for Spring 2022.

6.1 What The Students Said

We issued a short survey to the students over Summer, during their internship. Among a few other questions, we asked them whether they were using any of what they learned in class in the internship, what aspects of the class they valued the most, and what else they would like to see covered. One fourth of the students replied.

Many students reported having used strategies and techniques from the course to read and understand code in their internship. What they used varied and spanned nearly the entire set of topics. One student highlighted "I also took a look at unit tests to better understand the purpose of various functions" Another answered "PR etiquette and navigation knowledge". A third mentioned "I made diagrams during my internship so learning UML, etc really helped."

In terms of most important lessons, how to read code and patterns were the two most frequently mentioned. One of the students said "not being afraid to contribute to the code before understanding the whole code base", which aligns squarely with our overall goal of lessening the fear. Another mentioned "how much reading as opposed to writing is necessary" and a few others commented on UML. These answers align well with how the students ranked their personal improvement, with their ability to understand and work within the social context of a large-scale software system and ability to make a code contribution to a large-scale software system being seen as the areas where they advanced the most.

Many students mentioned that having only one lecture to become familiar with design patterns was insufficient (e.g., "More detail on design patterns for sure. We only went in depth on one or two, but I would like to go in depth about more", "I hope we could spend more time on the design pattern", "Design Patterns. I have been running into Design Pattern questions a TON during interviews"). They also commented that they would like to hear from guest speakers about how they have handled challenging situations in the past (e.g., "Bug fixing? I'd like to see some guest speaker talk about their experience in dealing with some high priority defect ticket under serious pressures"). Finally, they did not think the exams were particularly useful from a learning perspective (e.g., "Maybe exams could be optional or only for students who need extra credit").

From informally talking to the students, we learned of two other pervasive thoughts. First, they valued the introduction of the advanced IntelliJ functionality and plug-ins; many simply did not know and felt like they were now able to much more effectively work. Second, they did not like having to fill out the diaries each week. From an instructor perspective, however, these diaries allowed us to monitor how well individual teams did or did not work together, sometimes intervening. Moreover, what the students put down as new insights they learned from their activities allowed us to monitor whether the material resonated and was translated by the students into practice. We, thus, intend to keep the diaries.

6.2 What We Thought

Overall, we were much happier with this incarnation of the course as compared to its initial offering. Other than the complication that arose in week 9 regarding design principles and the use of a new design prompt, which necessitated bleeding a lot of material into week 10, the first eight weeks and the practical components of the course worked very well. To avoid the issue next time, we fully intend to no longer employ a midterm, spread out the design cycle and design principles materials over two weeks, and conclude the course with a proper lecture on history. We also envision revisiting the design patterns when we talk about the design principles, so that the students gain some more exposure. We realize the normal order is the reverse (principles first, then patterns), but given that we treat the topics from a reading to write perspective, patterns first and then principles fits better with the flow.

An important component was setting the expectations early and not low. The first offering of the course culminated in a voluntary submission of a pull request, but this did not give the students a good experience they can put on their resume. By moving topics around (particularly the social context lecture, which we moved much earlier), inserting an explicit step of selecting issues to work on, and making it clear from the beginning of the course that they would be making changes to the code base that would result in actual pull requests, we feel that the result was much more satisfying to the students. While one student commented "Repo owners are

extremely picky", we actually take that as an important lesson learned, because it will be no different in their future careers.

The community at large also served an important role. On the one hand, we had the voracious maintainer who addressed and closed all issues near instantly, leaving little for the students (we worked around this by asking the team to work on a proposed new feature of their own thinking instead) and some of the pull requests simply did not get any feedback, presumably because the the maintainers were busy with other, higher priority items. On one project the team submitted several pull requests which unfortunately became obsolete when a major new release came out a few days later that involved significant refactoring that eliminated the open issues altogether. At the same time, on a number of occasions it was clear that the maintainers took care of coaching the newcomers, encouraging them in their work, and giving them friendly feedback that often involved a "here is how to do this right". On a few occasions, too, teams received a congratulatory message on submitting their first pull request to a project. Overall, however, the diversity of experiences served as a rich base for the course at large – we could draw upon this diversity in teaching all students lessons they would not encounter in their own projects.

Another form of community served an equally important role: the community that has developed, curated, and published numerous valuable resources on which a course like ours can rely. Patterns, UML, and SOLID have a number of outstanding web sites dedicated to them, which made it possible for us to lay the foundations, but then redirect the students to learn more on their own, as they undoubtedly will need to do in their future careers as well.

The inclusion of guest speakers from industry enriched the course. While each highlighted different aspects of reading (and writing) code, their stories and experiences connected closely with the material that the students had been studying. The Google engineer performing an actual code review was a particularly strong addition, as was the exercise of reverse engineering the Lua software defined radio, which the guest speaker did by serving as the master teacher who enabled the students to perform the work under their guidance, were among the highlights. That said, a few of the guest speakers could benefit from organizing their message more clearly, perhaps with the help of slides. We will more proactively coach the guest speakers in that regard the next time around.

A particularly challenging issue is the selection of projects. We intentionally ensured that teams grouped around areas of interest, so that they would be more motivated to engage with the open source project of their choice. That said, there naturally exists a variability in the projects, the quality of their code, their documentation, their complexities hiding in the code, and their engagement with the community. While the TAs and instructor do take a look at the projects that teams want to study before they do so (e.g., to ensure sufficient churn, enough open issues, some basic level of quality, reasonably active community), this remains an inexact science and we were faced with a few surprises, as in the above. While this on the one hand enhanced the breadth of the educational experience for the class at large, it also means that some inequities exist in the challenges that different teams will face, as well as inequities in the potential for them to have their pull requests accepted towards the end of the quarter. We have not found a good way for us to account for this variability while still preserving the positives of a breadth of projects being worked on in class.

Finally, we continue to ponder the sequence of the lectures. While we feel we have arrived at a good order, there always is the desire to move an important topic from later earlier (e.g., move patterns earlier given how important they are in practice). That, however, always means another topic has to go later. Whether a better ordering exists than the one we have currently remains an unknown.

7 CONCLUSION

This experience report shared our design and delivery of a novel course that prepares students to read code in order to write code, together with reflections on what we felt went well and what could use improvement. The course design is directly influenced by the eight desired learning outcomes, though we note that it is a challenge to objectively assess the degree to which those learning outcomes have been met. Nonetheless, we believe the course is on the right track: students report using the strategies and techniques in their internships, they generally appear content with the materials of the course, they showed engagement in the in-class exercises, and they succeeded in making actual contributions to open source systems. That said, improvements can be made and we plan to do so as outlined in the above for the third offering of the course in Spring 2022. We hope this paper inspires others to offer similar kinds of courses as we believe a serious gap exists when it comes to graduates knowing how to read to write code. Course materials, including slides and assignments, are available upon request.

REFERENCES

[1] 2019. Master of Software Engineering (MSE). https://mswe.ics.uci.edu/
[2] 2020. APA Dictionary of Psychology. https://dictionary.apa.org/social-context
[3] Eric Allen, Robert Cartwright, and Charles Reis. 2003. Production programming in the classroom. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education (SIGCSE '03)*. Association for Computing Machinery, New York, NY, USA, 89–93. https://doi.org/10.1145/611892.611940
[4] Sogol Balali, Igor Steinmacher, Umayal Annamalai, Anita Sarma, and Marco Aurelio Gerosa. 2018. Newcomers' Barriers. . . Is That All? An Analysis of Mentors' and Newcomers' Barriers in OSS Projects. *Computer Supported Cooperative Work (CSCW)* 27, 3 (Dec. 2018), 679–714. https://doi.org/10.1007/s10606-018-9310-8
[5] Andrew Begel and Beth Simon. 2008. Novice software developers, all over again. In *Proceedings of the Fourth international Workshop on Computing Education Research (ICER '08)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1404520.1404522
[6] T. Busjahn and Carsten Schulte. 2013. The use of code reading in teaching programming. In *Koli Calling '13*. https://doi.org/10.1145/2526968.2526969
[7] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. 2011. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research (Koli Calling '11)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/2094131.2094133
[8] José P. Cambronero, Thurston H. Y. Dang, Nikos Vasilakis, Jiasi Shen, Jerry Wu, and Martin C. Rinard. 2019. Active learning for software engineering. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 62–78. https://doi.org/10.1145/3359591.3359732
[9] Cynthia L. Corritore and Susan Wiedenbeck. 2001. An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies* 54, 1 (Jan. 2001), 1–23. https://doi.org/10.1006/ijhc.2000.0423
[10] Mohsen Dorodchi, Erfan Al-Hossami, Mohammad Nagahisarchoghaei, Rohit Shenvi Diwadkar, and Aileen Benedict. 2019. Teaching an Undergraduate Software Engineering Course using Active Learning and Open Source Projects. In *2019 IEEE Frontiers in Education Conference (FIE)*. IEEE, Covington, KY, USA, 1–5. https://doi.org/10.1109/FIE43999.2019.9028517

[11] Alberto Espinosa, Robert Kraut, Javier Lerch, Sandra Slaughter, James Herbsleb, and Audris Mockus. 2001. Shared Mental Models and Coordination in Large-Scale, Distributed Software Development. *ICIS 2001 Proceedings* (Dec. 2001). https://aisel.aisnet.org/icis2001/64

[12] Scott D. Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. 2013. An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Transactions on Software Engineering and Methodology* 22, 2 (March 2013), 14:1–14:41. https://doi.org/10.1145/2430545.2430551

[13] Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. 2016. TASSAL: Autofolding for Source Code Summarization. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 649–652.

[14] Thomas Fritz and Sebastian Muller. 2016. Leveraging Biometric Data to Boost Software Developer Productivity. 66–77. https://doi.org/10.1109/SANER.2016.107

[15] Richard P. Gabriel. [n. d.]. Master of Fine Arts in Software. https://dreamsongs.com/MFASoftware.html

[16] Erich Gamma, Ralph Johnson, Richard Helm, Ralph E. Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Deutschland GmbH. Google-Books-ID: tmNNfSkfTlcC.

[17] Thomas B. Hilburn, Massood Towhidnejad, and Salamah Salamah. 2011. Read before you write. In *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE T)*. 371–380. https://doi.org/10.1109/CSEET.2011.5876108 ISSN: 2377-570X.

[18] Anna A Ivanova, Shashank Srikant, Yotaro Sueoka, Hope H Kean, Riva Dhamala, Una-May O'Reilly, Marina U Bers, and Evelina Fedorenko. 2020. Comprehension of computer code relies primarily on domain-general executive brain regions. *eLife* 9 (Dec. 2020), e58906. https://doi.org/10.7554/eLife.58906 Publisher: eLife Sciences Publications, Ltd.

[19] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05)*. Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/1101908.1101949

[20] An Ju, Hitesh Sajnani, Scot Kelly, and Kim Herzig. 2021. A case study of onboarding in software teams: Tasks and strategies. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 613–623.

[21] Zachary Karas, Andrew Jahn, Westley Weimer, and Yu Huang. 2021. Connecting the dots: rethinking the relationship between code and prose writing with functional connectivity. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 767–779. https://doi.org/10.1145/3468264.3468579

[22] Stephen King. 2010. *On writing: a memoir of the craft* (scribner trade paperback edition ed.). Scribner, New York.

[23] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering (ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 492–501. https://doi.org/10.1145/1134285.1134355

[24] Jochen Ludewig and Ivan Bogicevic. 2012. Teaching software engineering with projects. In *2012 First International Workshop on Software Engineering Education Based on Real-World Experiences (EduRex)*. 25–28. https://doi.org/10.1109/EduRex.2012.6225701

[25] Donis Marshall and John Bruno. 2009. *Solid Code*. Microsoft Press. Google-Books-ID: ZZtCAwAAQBAJ.

[26] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. 2019. How developers use API documentation: an observation study. *Communication Design Quarterly* 7, 2 (Aug. 2019), 40–49. https://doi.org/10.1145/3358931.3358937

[27] André N. Meyer, Earl T. Barr, Christian Bird, and Thomas Zimmermann. 2021. Today Was a Good Day: The Daily Life of Software Developers. *IEEE Transactions on Software Engineering* 47, 5 (May 2021), 863–880. https://doi.org/10.1109/TSE.2019.2904957 Conference Name: IEEE Transactions on Software Engineering.

[28] Tahmid Nabi, Kyle M. D. Sweeney, Sam Lichlyter, David Piorkowski, Chris Scaffidi, Margaret Burnett, and Scott D. Fleming. 2016. Putting information foraging theory to work: Community-based design patterns for programming tools. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 129–133. https://doi.org/10.1109/VLHCC.2016.7739675 ISSN: 1943-6106.

[29] Debora M. C. Nascimento, Christina F. G. Chavez, and Roberto A. Bittencourt. 2018. The Adoption of Open Source Projects in Engineering Education: A Real Software Development Experience. In *2018 IEEE Frontiers in Education Conference (FIE)*. 1–9. https://doi.org/10.1109/FIE.2018.8658908 ISSN: 2377-634X.

[30] Marian Petre and André van der Hoek. 2016. *Software Design Decoded: 66 Ways Experts Think*. MIT Press. Google-Books-ID: EVE4DQAAQBAJ.

[31] Wendy Pothier. 2021. Jamming Together: Concept Mapping in the Pandemic Classroom. *Ticker: The Academic Business Librarianship Review* 5, 2 (March 2021). https://doi.org/10.3998/ticker.16481003.0005.220

[32] Darrell R. Raymond. 1991. Reading source code. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research (CASCON '91)*. IBM Press, Toronto, Ontario, Canada, 3–16.

[33] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software?. In *2012 34th International Conference on Software Engineering (ICSE)*. 255–265. https://doi.org/10.1109/ICSE.2012.6227188 ISSN: 1558-1225.

[34] Janet Siegmund. 2016. Program Comprehension: Past, Present, and Future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. 13–20. https://doi.org/10.1109/SANER.2016.35

[35] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2008. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering* 34, 4 (July 2008), 434–451. https://doi.org/10.1109/TSE.2008.26 Conference Name: IEEE Transactions on Software Engineering.

[36] Therese Mary Smith, Robert McCartney, Swapna S. Gokhale, and Lisa C. Kaczmarczyk. 2014. Selecting open source software projects to teach software engineering. In *Proceedings of the 45th ACM technical symposium on Computer science education (SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 397–402. https://doi.org/10.1145/2538862.2538932

[37] Diomidis Spinellis. 2003. *Code Reading: The Open Source Perspective*. Addison-Wesley Professional. Google-Books-ID: 8lYbNfsAVT4C.

[38] Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. 2015. Social Barriers Faced by Newcomers Placing Their First Contribution in Open Source Software Projects. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW '15)*. Association for Computing Machinery, New York, NY, USA, 1379–1392. https://doi.org/10.1145/2675133.2675215

[39] Igor Steinmacher, Marco Aurélio Graciotto Silva, and Marco Aurelio Gerosa. 2014. Barriers Faced by Newcomers to Open Source Projects: A Systematic Review, Vol. 427. https://doi.org/10.1007/978-3-642-55128-4_21

[40] Arie Van Deursen, Maurício Aniche, Joop Aué, Rogier Slag, Michael De Jong, Alex Nederlof, and Eric Bouwers. 2017. A Collaborative Approach to Teaching Software Architecture. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 591–596. https://doi.org/10.1145/3017680.3017737

[41] Richard Wettel and Michele Lanza. 2008. CodeCity: 3D visualization of large-scale software. In *Companion of the 30th international conference on Software engineering (ICSE Companion '08)*. Association for Computing Machinery, New York, NY, USA, 921–922. https://doi.org/10.1145/1370175.1370188

[42] D. W. H. Wilmer, G. R. De Ridder, A. A. Kol, and D. C. Harkes. 2010. *Pacman*. Technical Report. https://repository.tudelft.nl/islandora/object/uuid%3A8bc685df-fcf4-4ed8-8ef3-a11e75f82c76