The Quest for a Practical Sophomore-Level Software Engineering Course

Roberto A. Flores
Department of Physics, Computer Science & Engineering
Christopher Newport University
roberto.flores@cnu.edu

Abstract

This paper describes our efforts starting since 2014 when we began developing a practical introductory sophomore-level software engineering course. The aim is to guide students into the fundamental theory and practice of building reliable software, with an emphasis on agile and object-oriented practices. Course topics revolve around three main themes: 1) managing complexity (how to model and scale software), 2) achieving quality (how to minimize defects) and 3) supporting usability (how to deliver user functionality). Students are exposed to theoretical and practical aspects of software production, such as software life-cycle models, strong-typing, testing, documentation, graphical user interfaces, UML, design patterns, version control systems and software deployment. The course is in constant evolution: near-future plans include adding build automation tools and DevOps concepts. We made the early decision to use reference materials available to our students at no cost; therefore, all reference materials are accessed online through resources afforded by our library.

1. Introduction

Traditional software engineering textbooks (such as Sommerville [1], Schach [2], Pressman [3] and Bruegge and Dutoit [4]) have existed for years in senior level or graduate student classes in computer science. These books offer comprehensive and thorough studies along with references to in-depth material. Their abstraction and reach, however, makes their content difficult for sophomore students who are starting their computer science studies and who need understanding of basic software engineering concepts in a practical setting.

Seeking a hands-on approach and relying on sources that bear no-cost to our students, we began developing a software engineering course to expose the practical aspects of (mostly agile) software engineering without neglecting theoretical concepts. We began offering this course in 2014 as a required course in our computer science curriculum. Its contents were inspired by our experiences teaching a major elective course (cross listed with our graduate program), which at the time was our only course exposing software engineering to our students (this course used Schach [2] and Bruegge and Dutoit [4] as textbook references). Our quest should not be misconstrued as quarreling against traditional textbooks (thanks to their unparalleled wisdom we now stand on the shoulders of giants) but as an empirical attempt to bridge students' passion for programming (which attracted many of us to computer science in the first place) with the broader study of software engineering. Our hope is that exposing students to techniques and practices readily relatable to their programming expertise will entice them to pursue deeper software engineering coursework.

The sophomore level course we designed is centered around three themes: 1) managing complexity (how to model and scale software), 2) achieving quality (how to minimize defects) and 3) supporting usability (how to deliver user functionality).

This paper is structured as follows: Section 2 gives a glimpse into the background of students taking the course and its place in our curriculum. Section 3 introduces the topics we cover in the course and how they fit within the complexity/quality/usability themes; and section 4 finishes the paper with conclusions and outlook.

2. Students background and curriculum

The intended audience of the course is second-year computer science majors, who typically have just ended a two-semester introduction to programming sequence (also known as CS1 and CS2). In these courses our students become familiarized with object-oriented programming in Java (using a procedural as opposed to an objects-first approach). They have not yet taken (but may take concurrently with software engineering) a data structures course. In particular, our students have prior knowledge on control statements (selection and iterative statements), arrays and lists, string handling and formatting, inheritance and polymorphism, exception handling, text file I/O, and the basics of socket programming, stacks, queues and linked lists.

URI: https://hdl.handle.net/10125/60203
ISBN: 978-0-9981331-2-6

The software engineering course is placed right at the point in which these programming concepts can be finessed, and their place highlighted within a production framework; and it provides a foundation for subsequent senior-level courses on open source development, design patterns, and mobile computing, which are part of the electives our students can take as part of their major requirements (other non-software engineering elective courses available include robotics, cryptography, databases and artificial intelligence). Due to our academic mandate, we teach this course in small-to-medium sized classes (usually 15 to 40 students per section) in computer-equipped classrooms (where each student/seat has a desktop computer). These settings allows us to use a mixed lecture-practicum approach, where we present theory in brief intervals combined with hands-on in-class exercises when appropriate.

3. Course topics

Figure 1 shows a concept map with the topics we cover in our introductory software engineering course. Different concepts are depicted as clear squares, and their practical components (if any) as black squares. For example, the concept of strong typing (shown in the lower right corner of the figure) uses enumerated and generic types as its practical components.

Moreover, the figure suggests how these topics are clustered across our three themes: from a bird's-eye view, complexity (#1) encompasses topics clustered on the lower third of the figure; quality (#2) by topics on the upper half; and usability (#3) by topics on the middle-right. To further exemplify this clustering, we can take the topic "Writing source code"–which supports complexity (#1) and is located in the lower part of the figure—can be handled by writing and reusing (object-oriented) source code that relies on strong typing, and tools to manage incremental versions, its deployment and documentation.

We proceed to describe below each of the three themes and their corresponding topics.

3.1. Supporting quality

Supporting quality deals with the issue of how to build software while minimizing its defects. More formally, Philip Crosby (as described in [5], p. 211, and in [6]) defined quality as conformance to specifications while preventing errors and having zero defects; or (in colloquial terms), quality software does what it is supposed to do (i.e., validation: finished product meets users' needs) and does it well (i.e., verification: the software being developed complies with specifications).

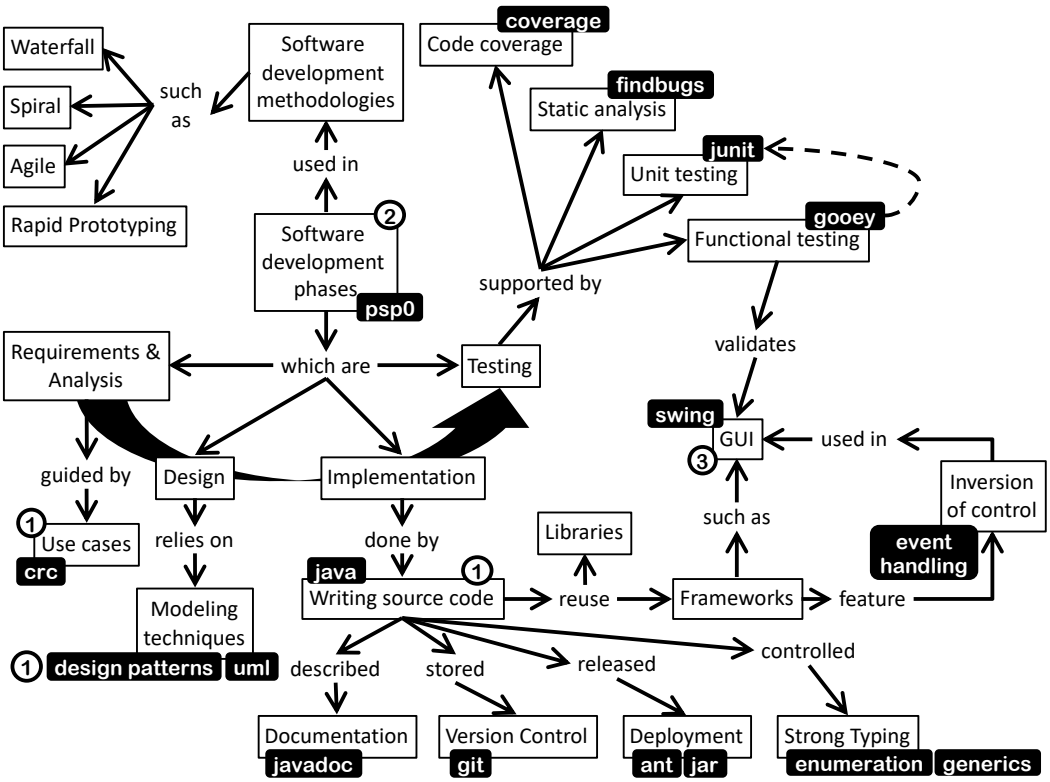

Figure 1. Concept map of course topics, their relationship and their distribution across themes: 1) managing complexity, 2) supporting quality and 3) supporting usability.

In terms of the course, we begin by describing prevailing software development phases (requirements & analysis, design, implementation and testing) as they are applied to sample software development methodologies (waterfall, rapid prototyping, spiral and agile) using [5] as main reference. It is worth mentioning that throughout the course we use the storyline "Today is your first day at an agile software engineering firm and you're asked to help with…" to highlight the relevance of topics and entice students to relate to course material. Following this line of thought, we challenge students (for example) to think not only on what program to write to satisfy user needs (functional requirements) but also on other constraints that may exist by use of computational media (non-functional requirements).

It is not uncommon of students that have just taken CS1/CS2 to see programming assignments mostly in terms of implementation: they are rarely given room for introspection in terms of development phases. This is a natural perspective given that their experience (in prior courses) was short programming assignments assessing expertise in programming constructs and library use.

To raise awareness of development phases and the actions students pursue when building software, we introduce Watts Humphrey's PSP0 [7] and provide a programming assignment as context to fill PSP0 forms. This is a multipronged exercise with two other goals in mind: foremost to engrain the importance of debugging and less so to reinforce practicing version control. On the one hand, we build on expertise from our CS1/CS2 courses, which require source code submission using GitLab [8] (we host our own servers to prevent public access to student work) to also require it in this course. On the other hand, we design this assignment to promote the use of visual debugging (a valuable yet underused tool) while minimizing the use of print out statements (an often abused and limited debugging technique). To this end, we provide students with JUnit tests that capture console output (of both out and error streams) and distribute them in an obfuscated JAR file (we use ProGuard [9] to inhibit disassembling with tools such as JD-GUI [10]). Although this arrangement is not bulletproof (e.g., students can still collect test data within tested classes and transcribe this data into a main method for separate testing) it creates a high enough burden that makes the alternative more appealing: using (and for a few student learning to use) a visual debugger. It is worth pointing out that this is the only assignment that dwells with PSP0, since students report that these forms tend to be tedious to handle when repeated in multiple projects.

As a cornerstone supporting quality coding, we place a particular emphasis on testing. The theoretical fundamentals we present include test types (glass vs black box testing), scope (e.g., unit, functional), purpose (e.g., acceptance, performance) and approaches (exhaustive, boundary, path analysis). Practical aspects of testing include code coverage, static analysis, and unit and functional testing, which we cover in assignments using EclEmma [11], Findbugs [12] (as a complement to Eclipse's compile settings), JUnit [13] and Gooey [14], respectively. As a side note, Gooey is an experimental homegrown JUnit-based tool that we developed to support functional testing of Java Swing applications (Thornton, et al., [15] offers a wider perspective on similar tools available to educators).

The second and third assignments in the course are designed with the intent of exposing students to writing comprehensive tests (alongside with a solution) to a given problem. The second assignment deals with object testing when abiding to equivalence relations on equality and comparability (including reflexivity, symmetry and, for the latter, transitivity). Together with the assignment description, students receive obfuscated instructor-made JUnit tests, which play the role of acceptance tests. A subsequent third assignment (later described in the section on supporting usability) deals with writing graphical user interfaces and their functional testing.

3.2. Managing complexity

Managing complexity deals with the issue of how to model and scale software as it is being developed. We divide this theme in two camps: analysis/design techniques, and implementation techniques and tools.

3.2.1. Analysis techniques. Analysis techniques act as a bridge to generate information conducive to an object-oriented design. As such we cover class-responsibility-collaboration (CRC) cards [16], which provide the initial understanding of actors and their relationships in the system, and use cases & user stories [17, 18], which detail interactions among actors in the process to achieve tangible results. This coverage is given within the framework of agile methodologies, using [18] as the basic reference. Modelling techniques are covered initially through abstraction, modularity and encapsulation concepts before introducing main UML diagrams (i.e., class, interaction and state diagrams) using [19] as reference. Subsequently, we cover a subset of design patterns [20, 21] that includes singleton, builder, abstract factory, command, decorator and iterator. The fifth (and last) assignment in the course revolves around an application implementing commands and redo & undo functionality with stacks; and depending on the topic it may contain an additional

design pattern, such as iterator (to parse content or generate infinite data sequences) or builder (to assemble an object at various stages).

3.2.2. Implementation techniques and tools. Implementation techniques and tools have a dual purpose: 1) to round up student programming expertise by introducing strong-typing techniques in the form of enumerated and generic types (which we cover beyond the basics: in the case of enumerated types by involving enumerated type fields; and in the case of generics by covering generics in classes and methods, in addition to inheritance wildcards); and 2) to present software development as a social activity supported by an integrated framework where documentation (to communicate usage and purpose), version control (to record incremental codebase evolution), and automated deployment (using Ant [22] and JAR files) offer a cohesive solution.

Out of the different topics we have chosen for the course, these (implementation techniques and tools) are the ones we are most eager to improve. We intend to include, expand, or outright replace build tool coverage (e.g., Gradle [23]), and continuous integration and delivery tools (e.g., Jenkins [24]) and concepts (e.g., DevOps [25]). As of today, we are still learning and planning which (and how) tools and practices can be adapted best to a classroom environment.

3.3. Supporting usability

Supporting usability deals with the issue of how to deliver user functionality. In its simplest form we introduce usability when discussing prototyping as a technique to address user interface design. We discuss usability primarily by exposing students to graphical user interfaces (GUI) programming, but we would like to explore this topic more broadly. A few references we might use to expand our understanding span from the technology agnostic [26], to the humorous [27] and pragmatic [28].

Our course uses Java Swing as the framework to program GUI programs. We did not arrive at this choice without controversy, since Swing does not fare well against the flair of web interfaces and rich client application frameworks. With the idea of remaining within the Java realm (students taking this course have a Java background) we explored JavaFX [29] as an alternative; however, its learning curve is substantially higher than with Swing and it can be dicey to teach to students who are yet to be exposed to any sort of event-driven and property-binding programming.

Our initial foray into this topic highlights the difference between libraries—which students are acquainted with (e.g., math, array list) and where user code retains the thread of execution between method calls—and frameworks—which use implicit invocation (also known as inversion of control) and where events are announced to those components registered for the event [30, p. 9].

We then divide our GUI coverage in two parts: structural (how to build a GUI using components, containers and layout managers) and behavioral (how to react to user actions using event-driven programming). The former includes basic containers (frames, dialogs and panels), components (buttons, labels, text fields, combo boxes and menus) and layouts (flow, grid and border). The latter introduces listeners (for components and windows) and anonymous classes. This initial coverage is supplemented with functional testing (Gooey), after which students are given their third assignment, in which they write a program and functionally test it based on provided use cases.

GUI coverage continues with a more eclectic set of functions (drawing shapes and fonts, displaying images, playing sound files, using timers and handling keyboard and mouse events) which build up to a fourth assignment on programming a 2D-based video game. As a side note: prior to releasing the assignment we ask students for their preference on a game to implement: out of the several game designs we have created in the past—which include minesweeper, pong and (a socket-based) battleship—students have strongly preferred space invaders. As such, we developed a sample program as guidance to students.

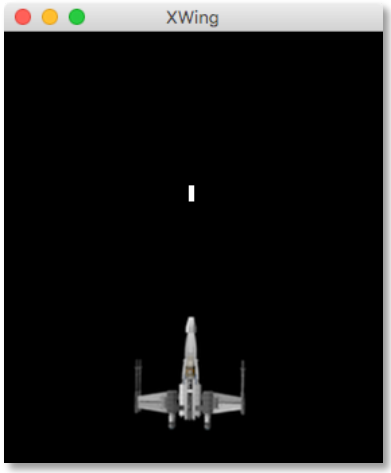

Figure 2. X-Wing program.

Figure 2 shows a snapshot of this sample program, which is coded in the classroom. The figure shows a window displaying a Star Wars X-Wing image, which reacts to keyboard arrows to move laterally, vertically and diagonally, and shoots (i.e., plays a shooting sound and displays a white rectangle representing a) photon torpedo when the space bar is pressed.

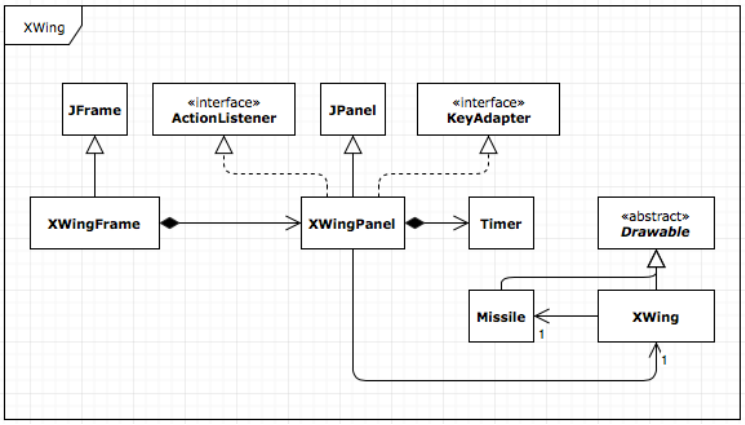

Figure 3. X-Wing UML class diagram.

Figure 3 shows the UML class diagram (drawn with an online tool [31]) used in the implementation of the X-Wing program. Execution of the program begins in the class XWingFrame, which displays a frame containing a panel that captures events from the keyboard (to receive input from the user) and timer (to drive the autonomy of the program) and holds a reference to an X-Wing object that manages a fired missile object. Figure 4 shows the UML class diagram of the space invaders program (a snapshot of which is shown in Figure 5). Comparing these diagrams helps students identify similarities between the sample program they wrote in class and the more complex program in the assignment. For example, both programs have a frame containing a panel with a timer; and both implement a common parent class (Drawable) from where all drawable (and movable) objects descend. However, even though the number of classes in both diagrams do not differ significantly (10 vs 16), the number of displayable objects between both is substantially higher (2 in X-Wing vs 57 in space invaders) and their interaction is more complex: on the one hand, the X-Wing program only requires tracking the missile while it travels within the frame's boundaries (at which point it gets reset); on the other hand, the space invaders program requires tracking missiles from both the base and invaders (up to four simultaneously), identifying intersects between missiles and targets (which signals a hit), alternating invaders images, and randomly generating the mystery ship and invader missiles, among others.

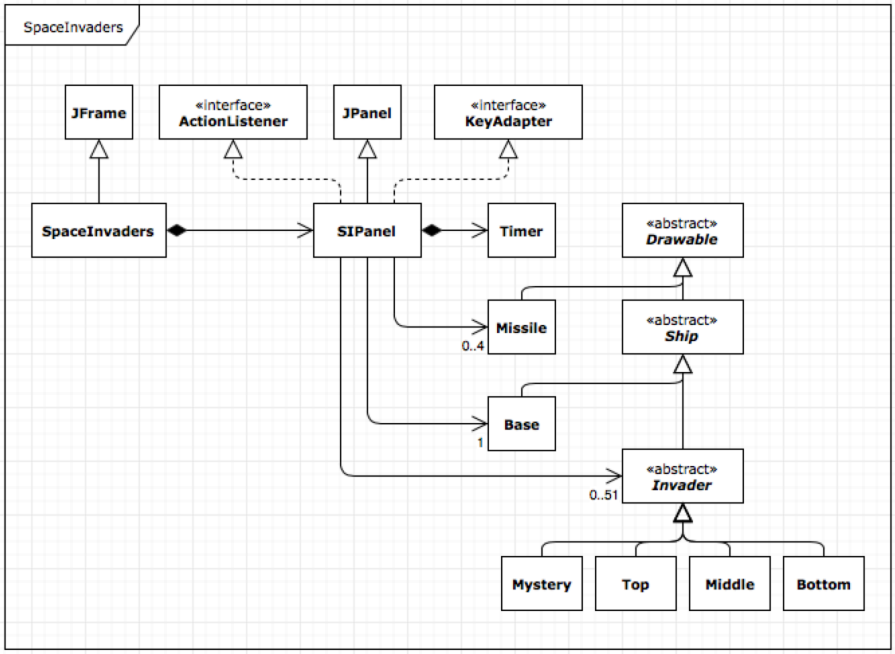

Figure 4. Space Invaders UML class diagram.

This assignment has proven to be more complex than any students have faced before, since it requires a sustained effort and attention to detail (as a side note: given its graphically-based nature, this assignment is the only one for which automated test cases do not exist).

Figure 5. Space Invaders program.

3.4. Course schedule

We would be remiss if we were not to provide some guidance on how we approach the course topics through a time line; in this case under a 13-week period session (i.e., a 14-week semester with one week for exams). As such Table 1 presents the approximate weekly order (solely based on our experience) in which topics are covered.

Table 1. Course topics under a weekly schedule.

Weeks	Theme	Topic
1-3	Complexity	Software development phases, PSP0, Software development processes, Agile methodologies (CRC, use cases), Java Documentation (Javadoc), Version control (git)
4-5	Quality	Testing principles (types, scope, purpose, approaches), Enumerated types, Generics, Unit testing (JUnit), Coverage (EclEmma), Static analysis (FindBugs)
6-7	Usability	GUI components (Swing), Event handling (anonymous classes, component & window listeners)
8-11	Quality	Functional testing (Gooey)
	Usability	GUI drawing, Event handling (mouse, keyboard, timer)
	Complexity	Software deployment (Ant)
12-13	Complexity	UML diagrams, Design patterns

4. Conclusions and future work

As famously remarked by Frederick Brooks in The Mythical Man Month [32] (no software engineering course worth its chops would omit at least a passing mention of this classic book): there is no silver bullet.

In this paper we provide a glimpse on our empirical attempts to design and build a pragmatic and engaging sophomore-level introductory software engineering course (skewed towards agile techniques). To this end, we used three themes (namely managing complexity, supporting quality and supporting usability) around which we build our course topics. We concur that our efforts can be improved in several ways not only by keeping up with technological advances but also by covering topics which we have neglected due to both time constraints within a semester and the limits of our own understanding and expertise. We pointed out some of these deficiencies throughout the paper, such as the need to improve coverage on build tools and continuous integration. Also, there are other topics we do not emphasize enough, such as working in teams (as a side note, students can opt-in to work with partners in assignments but most refrain citing conflicting schedules) or outright miss, such as project planning. Likewise, we have yet to do formal evaluation of the course's outcomes and mostly become aware of its impact on student education through anecdotal evidence from former students as they pursue summer internships or join the workforce upon graduation.

Nevertheless, by writing our experience in this paper we are aiming at engaging with the software engineering academic community to learn from their experiences and contribute (when possible) to identify pedagogical practices to better serve our students.

As a parting thought, we point out that all references used throughout the course (except for those we mention in Section 1) are accessed online by our students at no cost (an additional goal of our course design effort), either openly on the web or through our university library (using Safari books online [33]).

5. Acknowledgements

We are deeply grateful to the anonymous reviewers, whose thoughtful comments greatly helped us improve our paper. They made us aware of recent initiatives,

such as Software Engineering Methods and Theory (SEMAT) [34], which work to identify and share best practices with the software engineering community. We are also very grateful to Lynn Lambert, Mohammad Almalag and Ricardo Flores for their timely and helpful feedback to improve this paper.

6. References

[1] I. Sommerville, *Software Engineering*, 9th edition, Pearson, 2010.

[2] S.R. Schach, *Object-Oriented and Classical Software Engineering*, 8th edition, McGraw-Hill, 2011.

[3] R.S. Pressman, and B.R. Maxim, *Software Engineering: A Practitioner's Approach*, 8th edition, McGraw-Hill, 2014.

[4] B. Bruegge, and A.H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns and Java*, 3rd edition, Prentice Hall, 2009.

[5] F. Tsui, O. Karam, and B. Bernal, *Essentials of Software Engineering*, 4th edition, Jones & Bartlett Learning, 2017.

[6] P.B. Crosby, "Crosby's 14 Steps to Improvement", *Quality Progress*, American Society for Quality, December 2005, pp. 60-64.

[7] W.S. Humphrey, *PSP: A Self-Improvement Process for Software Engineers*, Addison-Wesley, 2005.

[8] GitLab. URL: http://gitlab.com

[9] Guardsquare, "ProGuard: The open source optimizer for Java bytecode". URL: https://guardsquare.com/en/products/proguard

[10] JD-GUI, "Java Decompiler: Yet another fast Java decompiler". URL: http://jd.benow.ca

[11] EclEmma, "Java Code Coverage for Eclipse". URL: https://www.eclemma.org

[12] FindBugs, "FindBugs: Find Bugs in Java Programs". URL: http://findbugs.sourceforge.net

[13] JUnit. URL: https://junit.org/

[14] Gooey, "Lean JUnit testing library for Java Swing applications.". URL: https://github.com/robertoaflores/Gooey

[15] M. Thornton, S.H. Edwards, R.P. Tan, and M.A. Pérez-Quiñones, "Supporting student-written tests of GUI programs", Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE), ACM Press, Portland OR, 2008, pp. 537-541.

[16] K. Beck, and W. Cunningham, "A laboratory for teaching object-oriented thinking", Proceedings of the conference on object-oriented programming systems, languages and applications (OOPSLA), New Orleans LA, 1989, pp. 1-6.

[17] I. Jacobson, "Object-oriented development in an industrial environment", Proceedings of the conference on object-oriented programming systems, languages and applications (OOPSLA), Orlando FL, 1987, pp. 183-191.

[18] K. Beck, and C. Andres, *Extreme Programming Explained: Embracing Change*, 2nd edition, Addison-Wesley, 2005.

[19] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd edition, Addison-Wesley, 2003.

[20] E. Gamma, and R. Helm, and R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[21] E. Freeman, and E. Robson, and B. Bates, and K. Sierra, *Head First Design Patterns*, 3rd edition, O'Reilly Media, Inc., 2004.

[22] S. Holzner, *Ant: The Definitive Guide*, 2nd edition, O'Reilly Media, Inc., 2005.

[23] T. Berglund, and M. McCullough, *Building and Testing with Gradle*, O'Reilly Media, Inc., 2011.

[24] B. Laster, *Jenkins 2: Up and Running*, O'Reilly Media, Inc., 2018.

[25] J. Humble, and G. Kim, and N. Forsgren, *Accelerate*, IT Revolution Press, 2018.

[26] W. Lidwell, and K. Holden, and J Butler, *Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design*, Rockport Publishers, 2010.

[27] S. Krug, *Don't Make Me Think, Revisited: A Common-Sense Approach to Web Usability*, 3rd edition, New Riders, 2013.

[28] J. Rubin, and D. Chisnell, and J. Spool, *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*, John Wiley & Sons, 2008.

[29] OpenJDK, "OpenJFX Project". URL: http://openjdk.java.net/projects/openjfx/

[30] D. Garlan, and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, Volume I, V.Ambriola and G.Tortora (editors), World Scientific Publishing Company, New Jersey, 1993. URL: http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf

[31] JGraph Ltd., "draw.io". URL: https://draw.io

[32] F.P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Anniversary edition, Addison-Wesley Professional, 1995.

[33] O'Reilly, "Safari". URL: https://safaribooksonline.com

[34] Software Engineering Methods and Theory, "Welcome - SEMAT". URL: http://semat.org/