Quality-Driven and Abstraction-Oriented Software Construction Course Design

To Fill the Gap between Programming and Software Engineering Courses

Zhongjie Wang, Hanchuan Xu, Ming Liu, Xiaofei Xu
{rainy,xhc,liuming1981,xiaofei}@hit.edu.cn
School of Computer Science and Technology, Harbin Institute of Technology
Harbin, Heilongjiang, China

ABSTRACT

Traditional Computer Science (CS) and Software Engineering (SE) curricula pay great attention to the training of programming skills and software engineering competence. In practice we find a large "gap" between the two perspectives: even for those students who have good programming skills, it is rather difficult for them to transform the programming-oriented thinking into the engineering-oriented thinking. Software construction, a key knowledge area (KA) in SWEBOK, plays a role for filling such gap in CS and SE education. We design "four transformations" in the course Software Construction and use multi-dimensional software artifacts as the index of the course contents, so as to train students on the design, programming and testing of a software system in terms of five key quality objectives. A set of gradually-deepening labs are designed for students to make practice on various quality-oriented software construction techniques. Two-year practice demonstrates that our course design significantly facilitates the transformation from programming-oriented training to engineering-oriented training and has been widely welcome by undergraduates from CS and SE.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**; • **Social and professional topics → Software engineering education**; **Model curricula**.

KEYWORDS

software construction, quality objectives, multi-dimensional software artifacts, software engineering, course design

ACM Reference Format:
Zhongjie Wang, Hanchuan Xu, Ming Liu, Xiaofei Xu. 2020. Quality-Driven and Abstraction-Oriented Software Construction Course Design: To Fill the Gap between Programming and Software Engineering Courses. In *ACM Turing Celebration Conference - China (ACM TURC'20), May 22–24, 2020, Hefei, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3393527.3393529

ACM TURC'20, May 22–24, 2020, Hefei, China

ACM ISBN 978-1-4503-7534-4/20/05…$15.00
https://doi.org/10.1145/3393527.3393529

1 INTRODUCTION

The term Software Construction refers to the detailed creation of working software through a combination of coding, verification, unit testing, integration testing, and debugging [3]. It is a core knowledge area (KA) in the Software Engineering Body of Knowledge (SWEBOK). In the process of developing a software system, software construction plays a "bridge" role between design activities and testing activities. It produces numerous configuration items (source files, documentation, test cases, and so on) that need to be managed in a software project, and among them executable code is the ultimate deliverables. The KA Software Construction is closely related to other KAs including Software Design, Software Testing, Software Configuration Management, and Software Quality [11].

In undergraduate curricula of Computer Science (CS) and Software Engineering (SE) disciplines, it is necessary to offer a "Software Construction" course in the second or the third year. Usually, CS/SE undergraduate students start from Programming Language courses (such as C, Python and Java) in their freshman year. By intensive programming training, students gain enough coding skills to implement small-scale functions [7]. Nevertheless, in order to grow into qualified software engineers, having only programming skills is not enough but they have to learn more software engineering courses such as Software Process and Modeling, Requirement Engineering, Software Design, Software Architecture, Software Testing, and so on [5]. It is worth noting that, programming-oriented thinking is quite different from engineering-oriented thinking that aims at complex problem solving because they have totally different emphasis [10]; in other words, these two perspectives cannot be naturally connected and transited in teaching, and there is a large "gap" between them. Even for those students who have gained good programming skills, it is rather difficult for them to transit rapidly from a good programmer into a good software engineer.

Software construction course is a good choice to fill such gap. Firstly, a majority of knowledge points in software construction are closely related to programming which students are very familiar with, therefore it is easy for them to be devoted into the course. Secondly, some engineering knowledge is gradually imported into the course and students are offered a whole new viewpoint on the implementation of software systems (i.e., a quality-oriented viewpoint), thus it makes a bridge to help students transit into engineering-oriented tasks such as design, modeling, testing and maintenance. In this way, students extend their vision to practical engineering activities and it lays a natural foundation for them to learn advanced software engineering courses in subsequent courses.

In Harbin Institute of Technology (HIT), from Spring 2018 we opened the Software Construction course in the second-year undergraduate curriculum. In this course, we try to fill the gap by four fine-grained transformations:

- From functionality-oriented programming to quality-oriented construction
- From concrete-oriented programming to abstract-oriented design/construction
- From narrow-sense programming to full-lifecycle development
- From manual programming to tool-aided development

After this course, students successfully transform their roles from programmers to developers: their working environment is extended from pure source code editors / compilers to modern software development toolkits including Integrated Development Environment (IDE), Software Configuration Management (SCM) tools, code review tools, unit test tools, and refactoring tools; they could develop medium-size software with above 5,000 lines of codes rather than small-piece trivial programs; they could design Abstract Data Types (ADTs) by the abstraction of multiple similar application scenarios and implement them via Object-Oriented Programming (OOP) so that codes can be effectively reused in a wide range [4]; they could make use of design patterns to give their design the ability to adapt to changes agilely; they could consider more quality metrics such as robustness, correctness, I/O performance, memory management performance, and so on. After two year's practice, we found students showed high enthusiasm on this course and gave it great recognition.

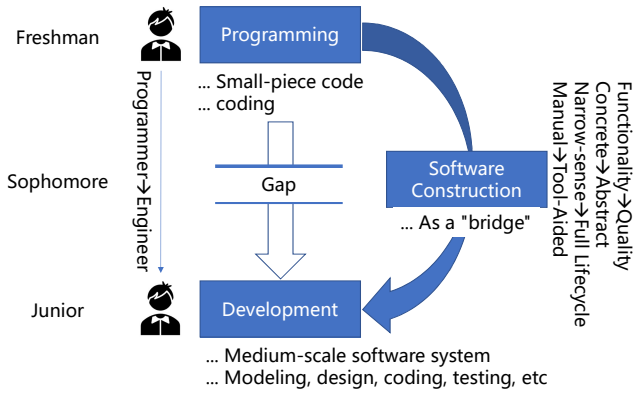

Figure 1: Software Construction Course: Bridging the Gap.

This paper gives a brief introduction on the design philosophy of this course, including the quality objectives covered by this course, multi-dimensional classification framework of software artifacts, and incremental and iterative labs design. Fundamental philosophy of this course is shown in Figure 1. As far as we know, until now only a few universities (both in China and worldwide) have opened this course (e.g., MIT, CMU, Nanjing University, Sichuan University), and we hope this paper could share some experiences with tutors of other universities.

2 GOALS AND PHILOSOPHY OF THE COURSE

The goal of this course is to help students understand both the building blocks and the design principles for construction of software systems, and then they could apply these principles to real-world software development.

Before enrolling into this course, students have already had enough knowledge and skills of writing programs according to simple specifications given by tutors; in this course, they are asked to focus on more software engineering activities, especially design (before programming) and testing (after programming). Given a specific software specification on both functionality and non-functionality (performance), there are infinitely many program-level solutions, and students are taught to answer the following questions before they go to detailed programming:

- What are the candidate design solutions in terms of the functional requirements in the specification?
- What are the differences between these variants of design?
- Which variant should you choose considering their performance differences?
- How can you synthesize a variant with desired non-functional quality attributes?

Using such an iterative design and programming process, students are trained to gradually form a new thinking pattern, i.e., before they do any programming even for the simplest task, they will consciously consider non-functional quality attributes and write down explicit design decisions; only after strict evaluation and comparison, are they allowed to proceed to programming. After intensive labs they will thoroughly throw away the "code-and fix" process used in their previous programming experiences.

The four fine-grained transformations is listed as follows.

2.1 From Functionality-Oriented Programming to Quality-Oriented Construction

This is a transformation on "development concerns", i.e., besides functional requirements in program specification, students should pay more attention to those non-functional ones [9], including understandability, reusability, extensibility, maintainability, robustness, correctness, performance, etc. Different design solutions have quite different quality exhibitions, and different software construction techniques will result in quite different quality exhibitions, too. After the course, students should be able to (1) identify the quality concerns from software specifications, (2) master typical software construction skills for each quality attribute, and (3) choose the best design and programming solution from multiple design candidates.

2.2 From Concrete-Oriented Programming to Abstract-Oriented Design/Construction

This is a transformation on "development objects", i.e., what kinds of software systems are to be designed and programmed. Before the course, students aim at developing a program for a concrete application scenario, but in this course, they have to switch to a more general scenario in which a set of similar but slightly different applications are to be developed. In order to decrease development cost and increase reusability/quality of software artifacts, abstraction is

a key design and construction technique [8]. After this course, students should be able to (1) design and implement abstract software artifacts based on similarity analysis and abstraction techniques such as Abstract Data Type (ADT) and Object-Oriented Programming (OOP) to make the program "cheap for develop"; (2) to extend and customize abstract artifacts to make them fit for different application scenarios, e.g., by Design Patterns; and (3) to evaluate the maintainability and extensibility of the design and implementation to make the program "ready for changes" and "easy to extend".

2.3 From Narrow-Sense Programming to Full-Lifecycle Development

This is a transformation on "development activities". Before the course, students pay most of their time to programming, but this is not enough to ensure the quality objectives. In this course, they are asked to work on other development activities include design, testing, configuration management (CM), team management, continuous integration, and so on. Although training on these activities are not too intensive due to limited teaching hours, after the course students should be able to (1) get to know the lifecycle of agile software development process; (2) pay attention to the satisfaction of quality attributes and make improvement in each of the development phase; and (3) have the consciousness of collaborating with others (e.g., to write elegant and beautiful code that is "easy to understand" by teammates).

2.4 From Manual Programming to Tool-Aided Development

This is a transformation on "development tools". It is a succession of the third transformation. It is difficult for developers to do so many development activities manually because manual development will result in low efficiency and poor quality. Tools have to be used, including static checking and dynamic checking tools, configuration management tools, unit testing tools, so that developers may automatically get comprehensive examination and objective measurement of the program, and further, to facilitate the continuous improvement of the program for making it "efficient to run" and "safe from bug".

3 QUALITY OBJECTIVES OF SOFTWARE CONSTRUCTION

Here we summarize five quality objectives covered by the Software Construction course. Actually, this course is organized in terms of the five quality objectives. Structure of the course is shown in Figure 2.

The five quality objectives are:

- Elegant and beautiful code → Easy to understand
- Low complexity → Ready for changes
- High robustness/correctness → Safe from bug
- Design for/with reuse → Cheap for develop
- High performance/efficiency → Efficient to run

The course 6.031 "Software Construction" in MIT [6] emphasizes specially on the objectives 1, 2 and 3, and the course 17-214 "Principles of Software Construction: Objects, Design, and Concurrency"

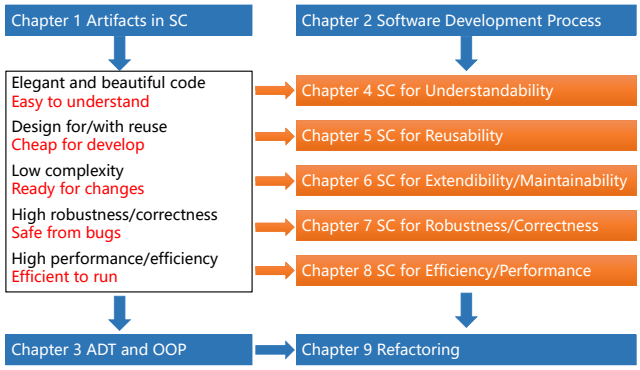

Figure 2: Five Key Quality Objectives and Course Organization Structure.

in CMU [2] focuses on the objectives 1, 2, 3 and 4. We extend the two reference courses to above-listed five objectives.

Our course is organized as follows. Chapter 1 is an overview by presenting a multi-dimensional classification framework of various types of software artifacts used or generated during software construction process (to be discussed in Section 4); Chapter 2 is a brief introduction to software development process including useful tools for each phase; Chapter 3 introduces ADT and OOP which are the core and fundamental part of the course and have profound effect on all of the quality objectives; Chapter 4-8 are software construction techniques for the five quality objectives, respectively, which are the main body of the course; and finally, Chapter 9 is a summary of the course by introducing program refactoring techniques based on those quality-oriented software construction techniques introduced in previous chapters.

4 MULTI-DIMENSIONAL CLASSIFICATION FRAMEWORK FOR SOFTWARE ARTIFACTS

Any software construction techniques act on specific software artifacts generated in the design, programming, testing and execution of software systems [1]. In order for students to understand and master these construction techniques, the first step of the course is to give a full view of these artifacts and connect them with closely-related quality objectives.

Figure 3 shows a multi-dimensional classification framework for software artifacts. It has three dimensions.

4.1 Build-Time and Run-Time Artifacts

The first dimension is about "time", i.e., what time a software artifact comes into being and is viewed and manipulated by developers. This dimension has two values: build-time and run-time. Build-time artifacts are used and generated during the development phase of a software system, and run-time ones are generated or used during the testing or execution phase.

For example, source code and its corresponding Abstract Syntax Tree (AST) are build-time artifacts, while a memory dump is a run-time artifact because only a program is loaded into memory during execution, can it be dumped out to a file.

	Moment		Period	
	Code-level	Component-level	Code-level	Component-level
Build-time	• Source code • AST • Interface-Class-Attribute-Method-Relations (UML Class Diagram)	• Package, file, library, static linking (UML Component Diagram) • Test case • Build script	• Code churn	• Configuration Item • Version
Run-time	• Code snapshot • Memory dump	• Package, library, dynamic linking, configuration, database, middleware, network, hardware (UML Deployment Diagram)	• Execution trace • UML Sequence Diagram	• Event log

Figure 3: Multi-Dimensional Software Artifacts Classification Framework.

4.2 Code-Level and Component-Level Artifacts

The second dimension is about "morphology", i.e., whether a software artifact is specific morphology of source code or the one of coarse-grained components.

Source code has many different forms, e.g., AST is a structural representation of source code, code churn is the modified code snippet in two neighboring versions, and UML Class Diagram is a logical representation of source code of an OO program.

The term "component" may refer to a file, a package, an external library, or an execution thread, or a physical machine where source code or other components are deployed.

4.3 Moment and Period Artifacts

The third dimension is about "lifetime", i.e., is an artifact generated and viewed in a specific moment or in a period of time?

For example, the source code is a "moment" artifact, while the code churn is a "period" one, because the former depicts the "picture" of specific time while the latter depicts the changes of source code in a period of time. Another example: memory dump is a "moment" artifact because it depicts the execution state of a program in memory, while an execution stack trace is a "period" artifact because each line of traces is recorded at different times, so the stack trace is generated in a period of time.

4.4 Eight Views in the Framework

The three dimensions divide the framework into eight views, which are explained in detail in Table 1, respectively.

4.5 Relating Software Construction Techniques to Quality Objectives in Eight Views

For each view in the framework, there are several software artifact types. Each quality objective mentioned in Section 3 is supported by a set of software construction techniques that act on different software artifacts in different views.

Taking the robustness/correctness as an example, effective software construction techniques that help improve program robustness and correctness appear in multiple views are listed below:

- In Build-time/Code-level/Moment view:
  - Error handling
  - Exception handling
  - Assertion
  - Defensive programming
  - Test-first programming
- In Build-time/Code-level/Period view:
  - Continuous Integration (CI)
  - Regression testing
- In Run-time/Code-level/Moment view:
  - Unit/integration testing
  - Debugging
  - Memory dumping
- In Run-time/Code-level/Period view:
  - Logging
  - Tracing

Construction techniques for all other quality objectives can be organized in this way, too. The advantage is obvious: students could easily keep in mind the types of software artifacts that each construction technique works on, and how this technique takes positive or negative effect on a specific quality objective.

Due to limited space, we cannot list all the construction techniques for all quality objectives and the views they cover respectively. Please refer to our course contents for more details.

5 LABS DESIGN

Although in this course we teach not only programming skills but also design and testing related skills, without doubts intensive programming exercises have strong effect on the learning outcome of students. A set of gradually deepening labs are designed for students to make practice on various quality-oriented software construction techniques.

In total there are five labs:

(1) Fundamental Java programming
(2) ADT and OOP
(3) Reusability and maintainability oriented programming
(4) Debugging, exception handling, and defensive programming
(5) Static and dynamic code analysis and code optimization

From the names of these labs people may think that each lab is focused on a set of construction techniques. The subtlety is that the programs developed in different labs are closely related with each other.

In Lab1, students are asked to do some basic Java programming on three simple problems. The first task is to write a function that judge whether an input file contains a magic square or not, and if the input file has illegal data format, the function should not exit directly but should give enough messages to remind the illegal type and where the "illegal" data appears in the file. In the second task, students are given a client program and are asked to write several classes to satisfy the client code. In the third task, students are asked to use a set of existing APIs to implement specific client demands. This lab has no any expectations on students to apply advanced construction techniques such as abstraction, exception handling, design patterns, and it is just a transitional step from previous programming courses to this course.

In Lab2, students need to fulfill two implementations for an ADT which is defined by a Java interface. Representations of the two implementations (Java classes) have been specified and students

Table 1: Eight View in the Framework

Views	Explanations
Build-time/Code-level/Moment	How are source code logically organized by basic program blocks such as functions, classes, methods, interfaces, and what are the dependencies among them? In other words, what does source code look like in a specific time? There are tree representations: lexical-oriented **source code**, syntax-oriented program structure (e.g., **Abstract Syntax Tree, AST**), and semantics-oriented program structure (e.g., **UML Class Diagram**).
Build-time/Component-level/Moment	How are source code physically organized by files, directories, packages, libraries, and the dependencies among them? What do these components look like in a specific time? **UML Component Diagram** is a typical exhibition of this view.
Build-time/Code-level/Period	How does source code evolve/change along with time? In SCM, it is described by **Code Churn**.
Build-time/Component-level/Period	How do files/packages/components/libraries change in a software system along with time? In SCM, it is described by **Software Configuration Items (SCI)** and their **versions**.
Run-time/Code-level/Moment	What does a program look like when it runs inside a target machine? How do programs behave in a specific time? This view focuses on variable-level execution states in the memory of a target computer, including **code snapshot** and **memory dump**.
Run-time/Component-level/Moment	How are software packages deployed into physical environment (OS, network, hardware, etc) and what are the files that the target machine needs to load into memory? **UML Deployment Diagram** is a typical representation of this view.
Run-time/Code-level/Period	How do programs behave along with time during the execution? For example:**UML Sequence Diagram** describes interactions among program units (objects) by specifying the calling sequence of multiple objects; **execution trace** involves a specialized use of logging to record information about a program's execution.
Run-time/Component-level/Period	How do software packages behave along with time during the execution? For example, **event logging** provides system administrators with information useful for diagnostics and auditing of various components of a software system.

only need to implement all the operations. Afterwards, students must use this ADT and its two implementations to develop two concrete applications (one is new, and another is from Lab1) to experience the "reuse" of ADTs in different scenarios and to compare the performance of two different ADT implementations. Note that implementing an ADT based on a given interface is not a real "design", but students do have the chance of involving themselves into a real ADT.

Lab3 is more complicated. Compared with Lab2 where students start from an existing ADT specification, in Lab3 no ADTs are available but they have to design and implement their own ADTs (including interfaces, representations, and implementations) from scratch. In order for elicitation, requirements of four similar applications are introduced in details, and students need to make use of abstraction techniques to design ADTs with high reusability and maintainability, i.e., "cheap for develop" and "ready for changes". Afterwards, four applications are implemented by these ADTs. This is the most difficult lab in the course.

Lab4 is a continuing step of Lab3. Students are asked to improve the robustness and correctness of the ADTs and the corresponding applications by techniques such as exception handling, assertion, defensive programming, and debugging. It is not necessary to develop many new source codes but to add pieces of auxiliary codes for robustness and correctness enhancement, i.e., "safe from bugs".

Lab5 is a continuing step of Lab4 for performance optimization. In previous labs, no performance expectations are raised but students pay attention to other quality objectives. Here they need to use static and dynamic checking tools to enhance the readability of source code (i.e., "easy to understand"), to identify the performance deficiencies, and then to eliminate them as far as possible (i.e., "efficient to run").

Structure and organization of the five labs are shown in Figure 4, in which orange rectangles and texts represent codes/specifications that have been given to students before the labs, and black ones are what students need to program.

6 REFLECTIONS AND FEEDBACK

After five labs, the LoC that students have developed is more than 5,000 or even to 10,000 (data is collected from their private GitHub[1] repositories which are managed by GitHub Classroom[2] service). This is an intensive training not only on programming but specially on quality-driven, tool-aided, abstraction-and-application-combined construction exercises. Students are asked to publicize their questions, feedback, and lessons learned from the course on a

[1]https://github.com
[2]https://classroom.github.com

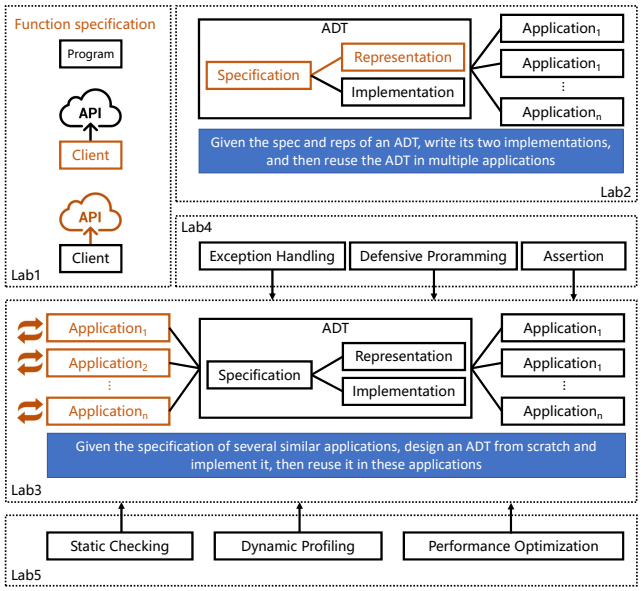

Figure 4: Structure and Organization of Five Labs.

QA platform (Piazza[3]) and personal blogs (mainly on CSDN[4]), and from these data we found about 70% students mentioned they have successfully transformed their thinking way on programming, and have got ready to welcome the challenge of "becoming a software engineer".

However, for other 30% students, this course is too hard for them to realize the transformations: they got unfavorable scores from labs and final exams. From their feedback we have identified the following possible reasons:

(1) The lack of sufficient programming skills and experiences impedes their capability of rapid prototyping and looking for better design/implementation solutions in limited time. As a consequence, they could only offer a primitive solution that lacks of detailed scrutiny.
(2) Abstraction-oriented programming is a great challenge for students who have well adapted to application-oriented programming scenarios. For example, in Lab3 students are asked to design an ADT that could fit for five different applications; most of students could easily implement the five applications separately, but to abstract them into highly reusable and customizable ADTs is a huge challenge.
(3) The lack of enough engineering experiences results in unconsciousness on the quality objectives or even the neglection of them. For example, in Lab3 students are asked to evaluate carefully their ADT design in terms of "ready for changes", but most of them cannot imagine what kinds of changes would happen in the future; even if tutors have already listed a set of potential changes, their intuition is not to improve their design but to "modify the codes when changes really happen".
(4) Construction techniques for those run-time views seem more challenging that those for build-time views. The reason is that the run-time states of a program are invisible and should be detected from the memory space by tools. This hinders them from catching the accurate states of run-time program and further, identifying wrong states and the corresponding cause roots in build-time artifacts.

These encourage us to continuously improving the course design. Hopefully 100% students would make a successful transformation.

[3]https://piazza.com
[4]https://blog.csdn.net

7 CONCLUSIONS

We introduce the design of our Software Construction course. In our Computer Science and Software Engineering curricula, this course plays a pivotal role of bridging programming courses and software engineering related courses, so as to promote the transformation of CS/SE students from programmers to software engineers. Features and advantages of the course design are succinctly listed as follows:

- Four transformations for the bridging;
- Five key quality objectives;
- Multi-dimensional classification framework on software artifacts;
- Classifying various software construction skills in terms of quality objectives, and mapping them to different views of software artifacts;
- Five gradually-deepening labs and intensive programming training.

ACKNOWLEDGMENTS

This course is supported by 2017 Education and Teaching Reform Project of Heilongjiang Provincial Education Department, China.

REFERENCES

[1] Sébastien Adam, Ghizlane El-Boussaidi, and Alain Abran. 2015. An approach for classifying design artifacts. In *2015 International Conference on Software Engineering and Knowledge Engineering*. 164–167.
[2] Jonathan Aldrich and Charlie Garrod. 2019. CMU Course 15-214: Principles of Software Construction: Objects, Design, and Concurrency. http://www.cs.cmu.edu/~aldrich/214.
[3] Pierre Bourque and Richard E Fairley. 2014. SWEBOK v3.0: Guide to the software engineering body of knowledge. *IEEE Computer Society* (2014), 1–335.
[4] William R Cook. 1990. Object-oriented programming versus abstract data types. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*. Springer, 151–178.
[5] Carlo Ghezzi and Dino Mandrioli. 2005. The challenges of software engineering education. In *International Conference on Software Engineering*. Springer, 115–127.
[6] Max Goldman and Rob Miller. 2019. MIT Course 6.031: Software Construction. http://web.mit.edu/6.031.
[7] Anabela Jesus Gomes, Alvaro Nuno Santos, and António José Mendes. 2012. A study on students' behaviours and attitudes towards learning to program. In *17th ACM Annual Conference on Innovation and Technology in Computer Science Education*. 132–137.
[8] Jeff Kramer. 2007. Is abstraction the key to computing? *Commun. ACM* 50, 4 (2007), 36–42.
[9] Sobia Mansoor, Arifa Bhutto, Neelma Bhatti, Noorulain aamir Patoli, and Mudassar Ahmed. 2017. Improvement of students abilities for quality of software through personal software process. In *2017 International Conference on Innovations in Electrical Engineering and Computational Technologies*. IEEE, 1–4.
[10] Greg Michaelson. 2015. Teaching programming with computational and informational thinking. *Journal of Pedagogic Development* 5 (2015), 51–65.
[11] François Robert, Alain Abran, and Pierre Bourque. 2002. A technical review of the software construction knowledge area in the SWEBOK guide. In *10th International Workshop on Software Technology and Engineering Practice*. IEEE, 36–42.