Teaching Object-Oriented Modeling as a Part of Programming Courses

Hidehiko Masuhara
Department of Mathematical and Computing Science
Tokyo Institute of Technology
Tokyo 152–8552, Japan
masuhara@acm.org

Abstract

Programming courses in computer science curricula usually teach the concepts and skills of programming, algorithms, data structures, and software development. Though the students who took those programming courses can solve programming exercises, they sometimes lack the problem-solving skills by programming. This essay describes the author's observations with the phenomenon and an attempt of teaching object-oriented modeling as a part of programming courses.

1 Solving programming exercises vs. solving problems by programming

This essay begins with the author's experiences with the computer science students who do not sufficiently have problem-solving skills by programming, even though they took the several courses on programming and became good enough to solve programming exercises. The students below were at the beginning phase of their bachelor/master research projects[1].

- A student wanted to add a new feature for an integrated development environment (IDE), which required to obtain all variables accessible from the current cursor position. Even though the student understood the concept of abstract syntax trees (ASTs), and was able to solve typical programming exercises on tree structures, he could not imagine how to do it for a real-world language like Java, which has many syntax rules introducing variables some of which complicate the scoping rules.

- Students were reading a textbook on programming language semantics. The book introduces the concept of variable environments followed by variable scopes and asks readers to consider design variations of variable environments. (A variable environment is a data structure that binds values of variables.) Even though the students understood the concept of variables scopes, and they can solve programming exercises with association lists or hash tables, it was not possible for them to think of several possible data structures and to discuss trade-offs.

- A student tried to simulate first-class delimited continuations (e.g., so-called shift/reset operators) by using a thread library. As it needs to change the behavior of expressions based on the number of executions at multiple program points and to pass information from one program point to another, it required to design a proper data structure that keeps track of all needed information. Even though the student can understand what the program should do, he had difficulties to design a proper data structure.

[1] Those projects are on design and implementation of programming language systems, which are due to the author's main research interests.

In: S. Chawla, B. Wadhwa, P. Muenchaisri (eds.): Proceedings of the 2nd Software Engineering Education Workshop (SEED 2019), Putrajaya, Malaysia, 02-Dec-2019, published at http://ceur-ws.org

Note that those students already had taken several programming courses (the curriculum is explained in Section 2.1) were able to *use* typical data structures like linked-lists and binary trees, and were able to solve basic programming problems (e.g., write a function to find a value in a binary sorted tree).

To summarize, students, even if they can use data structures and can solve basic programming problems, they can not design data structures and algorithms for solving larger problems. The author believes, though it is merely based on personal experiences, this problem would be a more general phenomenon to some extent.

2 Programming in computer science curricula

After noticed the problem in the previous section, the author started questioning what we are teaching; there might be something missing in our curriculum. This section first describes the curriculum at the author's institution, then examines standard curricula based on ACM/IEEE-CS 2013 report [Joi13]. Here, we only discuss courses related to programming among various topics on computer science.

2.1 Programming courses at the author's institution

The author's institution[2] provides the following courses related to programming.

- *Introduction to Computer Science* (2nd year) — the concept of programming, function definition, iterative computation, recursive functions, and list processing.
- *Data Structures and Algorithms* (2nd year) — the notion of data structures, linked lists, trees, various algorithms for data structures, and the notion of complexity.
- *Programming I* (2nd year) — the concept of abstract data types, recursive data structures, designing data structures, higher-order functions.
- *Computer Architecture* (2nd year) — Though the course is about computer hardware, students are required to build a hardware simulator in a programming language.
- *Programming II* (3rd year) — functional and object-oriented programming paradigms, semantics of programming languages, and interpreters and program transformation (in the old curriculum; we discuss this course in Section 4.)
- *Compilers* (3rd year) — Though the course is implementations of programming languages, students are required to build a small compiler in a programming language, which require to use various types of data structures such as ASTs and variable environments.

2.2 ACM/IEEE-CS 2013 curricula

The ACM and IEEE-Computer Society publish computer science curricula. Their 2013 report [Joi13] categorize programming courses in the *software development fundamentals* knowledge area. It assigns the area 43 hours Tier–1[3] courses, which is the longest among all the 18 knowledge areas in the curricula.

To quote from the report, it consists of the next four courses:

- Algorithms and Design — the concept and properties of algorithms, the role of algorithms in the problem-solving process, problem-solving strategies, and fundamental design concepts and principles.
- Fundamental Programming Concepts — basic syntax and semantics of a higher-level language, variables and primitive data types, expressions and assignments, simple I/O, conditional and iterative control structures, functions and parameter passing, and the concept of recursion.
- Fundamental Data Structures — arrays, records/structs, strings and string processing, abstract data types and their implementation, references and aliasing, linked lists, and strategies for choosing the appropriate data structure.

[2]It is the Department of Mathematical and Computing Science at Tokyo Institute of Technology. The department defines computer science and mathematics as the two core disciplines that every student is supposed to master. We, therefore, require students to take more mathematics courses than typical computer science departments. Our programming courses, however, are not largely different from typical computer science curricula.

[3]The tier–1 courses are the ones that should be covered by all students.

In: S. Chawla, B. Wadhwa, P. Muenchaisri (eds.): Proceedings of the 2nd Software Engineering Education Workshop (SEED 2019), Putrajaya, Malaysia, 02-Dec-2019, published at http://ceur-ws.org

- Development Methods — Program comprehension, program correctness, simple refactoring, modern programming environments, debugging strategies, and documentation and program style.

3 Skills to solve problems by programming

This section discusses skills to design and implement programs for larger problems, or to *solve problems by programming*.

3.1 Example steps of problem solving

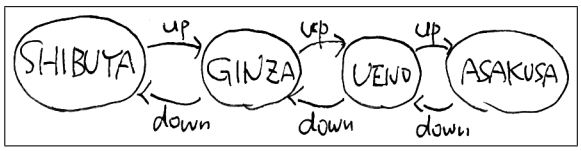
First, let us see how we solve such a problem by using the following example quoted from a textbook [FFF+12].

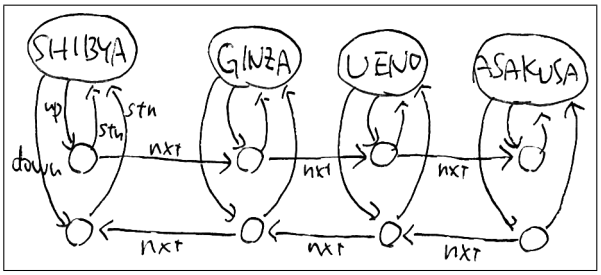
> **Exercise.** Many cities deploy information kiosks in subway stations to help tourists choose the correct train. At any station on a subway line, a tourist can enter the name of some destination; the kiosk responds with directions on how to get there from here.

We would solve this by taking the following four steps.

Elicit information that expresses the problem. Although there are many entities appear in the description, not all of them are needed for solving this problem (e.g., cities, information kiosks, and tourists). We would see that *stations*, *names of stations*, *destinations*, and *subway lines* are needed.

Design data structures for representing the data. We would come up with a couple of designs, the left one is a doubly-linked list of stations. The right one is an alternative that maintains a list of stations for each direction.





Design algorithms by using the data structures. To find the destination (either up or down) for a target station from an originating station, we would decompose the algorithm into (1) search for the target station upwards, and if not found, (2) search downwards. Then the search towards one direction is s a mere search on a linked-list.

Write a program that implements the data structures and the algorithms. For the left-hand side, we would define a station class (assuming in an object-oriented language) with a name, and up/down neighbor fields, and implement the search methods for two directions and the method to integrate the searches.

Note that those steps are often taken in a back and forth manner. For example, when we implement search procedures, we would need to write two almost identical procedures for two directions with the data structures on the left-hand side. We then may want to reconsider the data structure designs by comparing them with respect to modularity.

3.2 Object-oriented modeling

From the author's impression, many students have difficulties in the step of designing data structures. In this step, we need to decompose the information in the problem domain into smaller pieces and think about many possible ways of organizing those pieces of information. This step also requires to go back to the elicitation step to add or drop information, and to look ahead of applicable algorithms.

In this essay, we call this activity *object-oriented modeling*, though the term object-oriented modeling in a software development cycle refers to a rather smaller concept than what we are discussing as it is related to analysis of the problem and finding algorithms to solve the problem. Nevertheless, we can state our expectations to the student in this way.

In: S. Chawla, B. Wadhwa, P. Muenchaisri (eds.): Proceedings of the 2nd Software Engineering Education Workshop (SEED 2019), Putrajaya, Malaysia, 02-Dec-2019, published at http://ceur-ws.org

Students should be able to design right data structures by abstracting the problem domain and by considering algorithms to solve.

Note that this is not necessarily limited to object-oriented programming, but should be valid for other programming paradigms. This essay uses the term "object-oriented" modeling just because of the lack of appropriate terms in the other paradigms. We might call it "data modeling," but it should not be confused with data modeling for database designs.

3.3 Courses on software engineering

Some of the readers might wonder that the object-oriented modeling is taught as a part of the software engineering courses. However, software engineering is often positioned as advanced topics in many curricula. Moreover, even with a curriculum that contains software engineering courses, the abovementioned "object-oriented modeling" might not be intensively taught in those courses.

For example, the ACM/IEEE-CS 2013 curricula assigns 6 and 21 hours to the Tier–1 and 2 software engineering courses, respectively. The excerpted topics can be summarized as follows. As you can see, object-oriented modeling is merely introduced as one step of software development cycles.

- Tier–1: System design principles (e.g., levels of abstraction, separation of concerns) / Design paradigms (e.g., structured design, object-oriented analysis and design) / Structural and behavioral models of software designs / Design patterns
- Tier–2: Relationships between requirements and designs / Software architecture concepts and standard architectures / Refactoring designs using design patterns / The use of components in design

We also notice that many of the concepts in software engineering are related to the abovementioned "object-oriented modeling." By exercising object-oriented modeling, we would be able to give good introductions to many concepts in software engineering such as modularity (separations of concerns), design patterns, and reusability.

4 Teaching object-oriented modeling as a part of programming courses

Since 2015, we started teaching object-oriented modeling as a part of programming courses. This section describes the design of the course, which corresponds to "Programming II" in the list in Section 2.1. The main goal of the course is to train students to design choices of class hierarchies by analyzing the problem descriptions.

4.1 Textbook: how to design classes

We design the course by following a textbook draft titled "How to Design Classes" [FFF+12]. The textbook teaches a process of (1) analyzing the problem by reading the problem description with eliciting relevant data elements and example cases, (2) drawing class diagrams based on the analysis, (3) create test cases, (4) implement classes, and (5) run tests.

It begins by creating a small and simple fragment of software systems, such as "design a data structure for recording sales of a coffee shop", and growing to data with nested structures, behaviors of those data, recursive data structures and methods, subclassing, generic data structures, and frameworks and design patterns.

The textbook follows the style of the textbook entitled "How to Design Programs" [FFFK01, FFFK18]. Both advocate the use of design recipes that guide the students to follow in the problem-solving steps.

4.2 Teaching style

We incorporate some ideas from a flipped classroom [Fla13]. The class is scheduled for seven weeks, each has three 90 minutes time slots. Those slots in a week are divided into (1) a lecture given by the instructor, (2) a small set of follow-up exercises, and (3) a group laboratory to solve more challenging exercises.

The third slot is organized in this way: one randomly chosen student is appointed to lead the classroom to solve an exercise. He/she is responsible to ask opinions of the other students and write down the agreed design on the blackboard or implementation on the projected computer screen. The instructor observes the course of the discussion, gives hints and comes back to the points in the lecture by generalizing the lessons learned from the exercises.

In: S. Chawla, B. Wadhwa, P. Muenchaisri (eds.): Proceedings of the 2nd Software Engineering Education Workshop (SEED 2019), Putrajaya, Malaysia, 02-Dec-2019, published at http://ceur-ws.org

4.3 Reactions

Since we have not carried out systematic assessments, the author can only report subjective impressions of the students who took the course.

- We often saw heated discussion on possible class designs. For example, when the students discussed the exercise shown in Section 3.1, several students were insisting on the advantages of their designs.
- The majority of students became able to class design problems involving four to six classes in their final examination of the course.
- From the course evaluation by students, the majority of them liked the topics and style of the course.

It is however not clear to the authors whether the students mastered the skills and can apply to the software developments in their projects late years.

5 Concluding remarks

This essay pointed out that the ability to solve problems by programming is hard to master for many students, and introduced the author's attempt to teach by focusing on object-oriented modeling in the programming courses.

There would be further opportunities for improvements.

- Reconsider the balance between lectures and exercise so that students can solve more exercises.
- Asses the problem-solving skills before and after the courses.
- Incorporate team development for understanding the importance of interface designs.
- Incorporate evolution processes for understanding the importance of modularity.

References

[FFF+12] Matthias Felleisen, Matthew Flatt, Robert Bruce Findler, Kathryn E. Gray, Shriram Krishnamurthi, and Viera K. Proulx. *How to Design Classes*. unpublished draft, June 2012. `https://felleisen.org/matthias/HtDC/htdc.pdf`.

[FFFK01] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: an Introduction to Programming and Computing*. MIT Press, 2001.

[FFFK18] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs, Second Edition: an Introduction to Programming and Computing*. MIT Press, 2018.

[Fla13] Matthew Flatt. I flipped a class, and I liked it. `http://unclosedparenthesis.blogspot.com/2013/12/`, December 2013.

[Joi13] Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, New York, NY, USA, 2013.

In: S. Chawla, B. Wadhwa, P. Muenchaisri (eds.): Proceedings of the 2nd Software Engineering Education Workshop (SEED 2019), Putrajaya, Malaysia, 02-Dec-2019, published at http://ceur-ws.org