Introducing Agile Practices in Object-Oriented Programming: Applying *How To Solve It* Patterns

Yu Chin Cheng, Taipei Tech
Kai H. Chang, Auburn University

We report the experience of our effort to include numerous agile practices alongside the C++ language features and object orientation in teaching object-oriented programming, the second programming course in the imperative-first paradigm of CC2001 adopted at our institutions. We use long running examples that solve problems of sufficient complexity in class and build assignments as their extensions. The problem solving heuristics *How To Solve It* guide the analysis, design, coding, and review activities both in instructor's preparation and in class. In preparing a problem, the instructor records problem decomposition as an AND-OR graph. The instructor then identifies learning opportunities and explores how best to order the constituent subproblems for instruction. Coding is driven by tests and is performed on the only computer in the classroom. Pair/mob programming engages all students in problem solving and coding and is supplemented by occasional instructor demonstrations. Exit surveys taken by students of two recent course offerings show encouraging results.

Categories and Subject Descriptors: H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Evaluation/ methodology*; H.1.2 [**Models and Principles**]: User/Machine Systems—*Human Information Processing*; I.5.1 [**Pattern Recognition**]: Models—*Neural Nets*

General Terms: Human Factors

Additional Key Words and Phrases: object-oriented programming as a second course, problem solving heuristics, agile practices

ACM Reference Format:

Cheng, Y. C. and Chang, K. H. 2019. Introducing Agile Practices in Object-Oriented Programming: Applying *How To Solve It* Patterns. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 9 (March 2020), 9 pages.

1. INTRODUCTION

Programming courses offer the first opportunities for students to acquire programming knowledge and software development skills that carry them through their computing curricula and into professional careers. In the case of object-oriented programming in C++ as a second programming course (CS112) in the imperative-first paradigm of CC2001 [1], the skill set typically includes mastering language features, standard libraries, and object orientation. Due to the widespread adoption of agile development, we believe that the skill set should also factor in selected agile practices including *iterative and incremental development (IID)* [2], *test-driven development (TDD)* [3], *refactoring* [4], *pair programming* [5], *mob programming* [6], *continuous integration (CI)* [7], and so on. These practices are summarized in Table 1.

To weave the three main threads into a wholesome course, we have experimented with solving sufficiently complex problems in class. In addition to providing a rich context for learning language features and object-oriented design, the solution of a problem typically spans multiple class meetings to provide a working context for IID. In the

Author's address: Y. C. Cheng, Department of Computer Science and Information Engineering, Taipei Tech, Taipei 10608, Taiwan; email: yccheng@csie.ntut.edu.tw; Kai Chang, 3101 Shelby Center, Auburn, AL 36849; email: changka@auburn.edu

past few years, we have applied *How To Solve It*, the well-known heuristics for mathematical problem solving [8], as the main framework for instructional planning and execution. A description of *How To Solve It* and its four steps for teaching and learning object-oriented programming - *understanding the problem, devising a plan, carrying out the plan,* and *looking back* - can be found in [9], which was subsequently expanded into a pattern language in [10]. *How To Solve It* can be easily mapped to an agile method. In the case of Scrum [11], *understanding the problem* occurs during *product backlog development and refinement*; *devising a plan* maps to *sprint planning*; *carrying out the plan* maps to the main development work taking place between *sprint planning* and *sprint review*; and *looking back* maps to *sprint review* and *sprint retrospective*.

In applying *How To Solve It* on solving complex problems in class, the bulk of time is spent on live coding by the instructor and by students in the third step of *carrying out the plan*. This means that there are lots of opportunities for things to go wrong and the instructor needs to be able to pull things back on the right track. Furthermore, the students will need timely opportunities to practice outside of class, ideally building on what they have already learned in class. The former is solved by organizing and sequencing the problem components with an AND-OR graph [12] in the instructor's pre-class preparation and by applying test-driven development [3] during in-class coding that engages the whole class in mob programming [6]. In the latter, programming assignments that build on the example in class are given, and a continuous integration system [7] is set up to allow the students to submit their programs and know the grade instantly. As a result, agile practices including unit testing, test-driven development,

The rest of this paper describes the experience we gained so far. Section 2 details the problem domain selection, the decomposition into subproblems using the AND-OR graph, and the sequencing of the subproblems for instruction. Section 3 describes how in-class coding evolved from instructor demonstrating to pair programming and mob programming. Section 4 describes how continuous integration is included in homework submission loop. Section 5 gives a summary.

2. PROBLEM DECOMPOSITION AND SEQUENCING

Both in pre-class preparation by instructor and in classroom problem solving and coding with students, a complex programming problem is decomposed into subproblems, which are then properly sequenced. Decomposition takes place in the first step of *understanding the problem*. Decomposition enables the instructor to grasp the whole picture of the problem and to explore the learning opportunities pertaining to language features, agile practices, and design concepts embedded in the problem. Sequencing takes place in the second step of *devising a plan* and determines the order of the subproblems to tackle.

In this section, we shall illustrate problem decomposition with the AND-OR graph, a well-known construct for problem solving in artificial intelligence [12]. In brief, a node in the AND-OR graph represents a problem to solve. An AND-node represents a problem decomposed into a number subproblems, *all* of which must be solved in order to solve the problem; an OR-node represents a problem decomposed into a number of subproblems, among which *only one* must be solved. A *terminal* node represents a problem that is trivial enough to be solved directly without further decomposition.

The problem of computing the perimeter and the area of a convex polygon is used as an example; the solution is implemented in the C++ language.

Problem P. *Compute the perimeter and area of a convex polygon given its vertices.*

Problem P is the third in a series of related problems used in our object-oriented programming course [10]. It is preceded by the problems of computing the inner product of two vectors and of solving a system of linear equations with the Gauss-Jordan method. By the time problem P is attempted, code as well as tests have been written for the classes `Vector` and `Matrix`. We want the two classes and the tests to be reused, and enhanced

Table I. Agile practices included in the object-oriented programming course

Practice	Purpose
How To Solve It	solving a problem iteratively and incrementally
test-driven development	driving design of classes and functions; defining "done coding a method"; enabling code improvement
refactoring	changing implementation for improved design and cleaning code
pair/mob programming	engaging students to think and code in solving problems in class
continuous integration	making unit tests useful; encouraging multiple rounds of improvements in homework

where necessary, in the solution of problem P. The domain of the problems is appropriate since the students have already taken or are simultaneously taking a course in linear algebra. The selection ensures that the problems to solve are within the students' grasps while still being complex enough, thus providing a rich enough context for students to learn the skills but without grappling with extra difficulties brought by an unfamiliar domain.

2.1 Problem decomposition

Problem P can be obviously decomposed into three conjunctive principal parts as shown in Figure 3: representation of a convex polygon (problem P_1), computation of the perimeter (problem P_2), and computation of the area (problem P_3). Further analysis leads to the strategies for computing the perimeter and the area[1] as illustrated in Figure 2. In Figure 2(a), the perimeter is calculated by traversing the sides in the counter-clockwise (or clockwise) order, accumulating the lengths of the sides along the way. The area is calculated by decomposing the convex polygons into triangles and accumulating their areas. As illustrated by Figure 2(b), both computations depend on keeping the vertices of the convex polygon correctly ordered.

Further decomposition of a subproblem is performed when appropriate. In Figure 3, problem P_1 of representing convex polygon is further decomposed into two subproblems, problem P_{11} of constructing a convex polygon assuming that the given vertices are ordered correctly, and problem P_{12} of ensuring that the vertices are correctly ordered before the polygon is created. In particular, solving problem P_{12} involves sorting the vertices in some way; Figure 2(c) shows one such way. Here, the instructor could choose to implement a sorting function for ordering the vertices; or, she/he could take this opportunity to introduce the sorting function from the standard template library. The latter option is attractive since learning to use the application programming interface in the standard template library is essential for programming in C++. Also, since sorting requires the use of a comparator, problem P_{12} of ordering vertices is further decomposed into two conjunctive subproblems, the *learning* problem P_{121} of practicing the use of `std::sort` and problem P_{122} for implementing the custom comparator for use by `std::sort`.

The learning problem P_{121} involves any of the three alternative ways of supplying a comparator: with a pointer to a comparator function, with a comparator object that overloads the call operator taking two arguments for comparison[2], or with a lambda function[3]. Thus, P_{121} is an OR-node with three children P_{1211}, P_{1212} and P_{1213}, but the instructor may want to solve all three learning problems, anyway, to familiarize the students with the three ways of interoperating with `std::sort`. Once the students learn how to work with `std::sort`, problem P_{122} of defining a comparator for sorting vertices is tackled. Note that only function object and lambda function work because sorting the vertices according to Figure 2(c) depends on the centroid and the reference vector being made available for each invocation of the comparator.

When all terminal nodes in the AND-OR graph are trivial enough, decomposition stops.

[1] `http://htsicpp.blogspot.tw/2014/10/convex-polygon.html`
[2] `http://www.cplusplus.com/reference/algorithm/sort/`
[3] `http://www.cprogramming.com/c++11/c++11-lambda-closures.html`

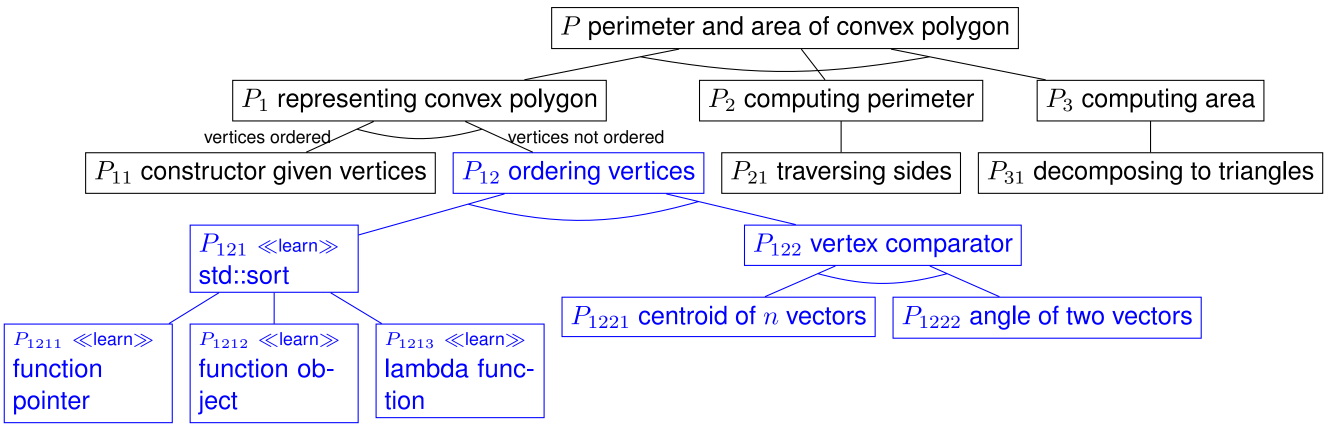
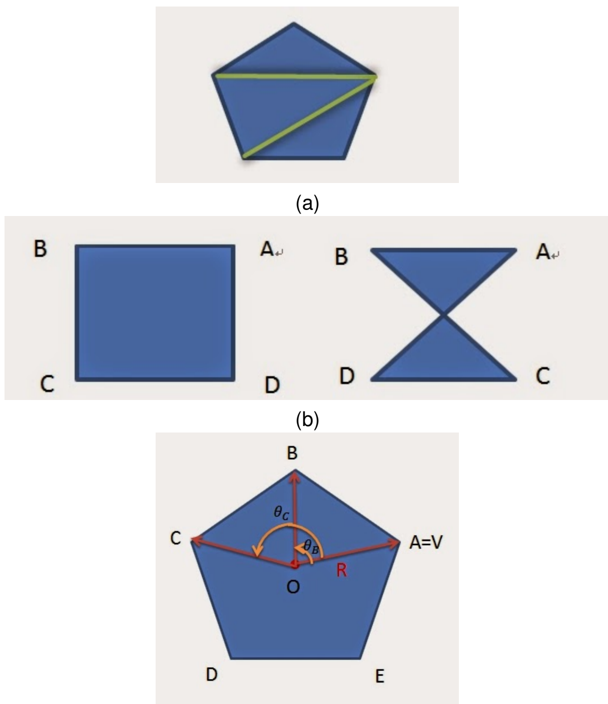


Fig. 1. An AND-OR graph decomposition of the problem of computing the perimeter and the area of a convex polygon. The subtrees in black and blue are the problems factored in the first and the second increments, respectively.



(a)

(b)

(c)

Fig. 2. (a) Computational strategies for calculating perimeter and area of a convex polygon. (b) For the strategies to work, vertices of the convex polygon must be ordered correctly. (c) Ordering the vertices counter-clockwise around the centroid of the convex polygon with respect to the reference vector R from the centroid to a designated vertex A.

2.2 Sequencing the subproblems

With the AND-OR graph of problem P obtained, the instructor next considers the order to proceed solving the subproblems. There are abundant traversal strategies to choose from, e.g., solving the subproblems in the order they are encountered with the depth-first traversal, beginning at the terminal nodes in a bottom-up style, and so on.

To emulate IID, subsets of the subproblems in the solution tree that constitute *increments* toward the final solution of problem P are identified, where the outcome of an increment reveals a whole picture of the solution to the problem as much as possible. In particular, a working program is ready for review at the end of each increment. In Figure 3, subproblems P_{11}, P_2 and P_3 are selected as the solution subtree for the first increment, which amounts to solving the problem P *assuming that the given vertices are correctly ordered*. Note that P_{11} subsumes P_1 with the simplifying assumption. In the second increment, subproblem P_{12} of creating a convex polygon is solved by first correctly ordering vertices (the subtree in blue). In so doing, the first increment can quickly generate a working program that can be executed and reviewed, thus giving the students a whole picture in the solution of the problem. The second increment tackles subproblem P_{12}, which is algorithmic in nature and involves learning the sorting functions in the standard template library. If problem P_{12} is tackled right after tackling problem P_{11} of the representation of convex polygon, calculations of the perimeter and the area are held back, not allowing a working program to be created as quickly as possible.

The sequencing in Figure 3 also makes a homework assignment as an extension of the code in class possible. For example, in a fast-paced rendition of problem P, the first increment is done in class while the second increment is assigned as homework.

3. IN-CLASS CODING WITH TEST-DRIVEN DEVELOPMENT AND MOB PROGRAMMING

In-class coding constitutes the largest portion of the classroom activities, taking place during the step *carrying out the plan*. In a nutshell, the next task from the sequenced AND-OR graph is tackled. A task can be solving a leaf-level subproblem, integrating solved subproblems of an internal node, or refactoring. Adopting TDD, for the task on hand, a test case is created or an existing test case is used. Inside the test case, one test is implemented at a time, and then the corresponding code to make *all* unit test cases pass is written. The process is repeated until the the task is completed.

The instructor then moves on to the next task, and so on. At the end of every class meeting, the code is checked in to a public repository (in the most recent case, github[4]). The students download the code for review and further exploration.

Since in-class coding could potentially involve any of the students, a disciplined style of work is needed to bring consistency. Over the last few years, our in-class coding has evolved from instructor demonstrating to pair programming [5] and then to mob programming [6]. Before 2013, the instructor wrote all the code in class. While the style is popular for online courses on programming, there is the question of how much the students learned from watching demonstration. To borrow the Scrum phraseology, the students played *chickens* instead of *pigs* during class coding: they are *involved* (they needed to do assignments which use and extend the code written in class) but not *committed* (they did not code in class). Two shortcomings were observed. First, since the students did not code in class, the instructor was missing out on learning in real-time the impediments they had - whether it was a new language feature that is difficult to grasp or an old one that they have learned but forgot about. Second, lacking action, some students easily zoned out or wandered off to other distractions.

Beginning in 2014, in addition to instructor demonstrating coding, we invited the students - one at a time - to code in class. Knowing that some students would have difficulties, not to overburden them to face the difficulties alone, we used pair programming [5]. The student volunteer played the role of *driver* to control the keyboard and the mouse; the instructor played the *navigator*, giving instructions to the student driver. The instructor avoided micro-instructing the student driver. For example, an instruction such as “go through all elements of the container”

[4] `https://github.com/yccheng66/oop2017s`

is preferred over spelling out code of a for-loop. All the other students observed while the coding was going on. Pair programming allowed the instructor to learn, first hand, if the student driver had difficulties understanding the instructions or turning the instructions into code. By asking the other students, the instructor could quickly determine if the student driver's capability was representative of the students in class. If so, the instructor should quickly whip up additional simple but related exercises to bridge the gap. The student driver changed hands at the completion of every task.

Beginning in 2016, student participation further evolved into a style close to mob programming [6]. In a way, this was anticipated. In pair programming, the instructor would ask the students for their suggestions when the driver got stuck. So naturally, the interaction evolved into mob programming: while one student played the driver role to control keyboard and mouse, *all the other students played the role of navigators* who shouted out instructions to the driver for the coding task on hand. Free from navigating most of the time, the instructor focused on facilitating the process. Often, there were competing alternatives from the student navigators. The instructor could take the opportunity to help the students to explore the competing alternatives - implementing some of them along the way - and help them pick the best one, or challenge the students to come up with a better alternative when all of the alternatives were not good enough.

Here's an example of mob programming taking place in class in the test-driven style. When an individual test of a test case is implemented for the first time, it usually involves the following actions: preparing test data, creating an object, setting the object into a correct state, calling a method of the object, and checking the result. In this sequence, not only the constructor and the method to achieve the task is written, but also the supporting member functions such as getters and setters are implemented *only* as needed. Figure 3 shows the one unit test written for the task P_{11}. The instructor could set the context by asking "what is an example of a minimal, non-degenerate convex polygon?" The navigator would very likely answer "a triangle", and draw up a triangle with three vertices $(0,0)$, $(3,0)$, and $(3,4)$ on the whiteboard. The navigator then guides the driver to write down the constructor and well as the assertion for checking that the triangle is created with the desired data members. On the receiving end, the driver prepares arrays and turns them into instances of `Vector`, a previously written class. Up to this point, the driver compiles and runs the test under construction to make sure that all are correct. The driver then pass the array of Vector with number of vertices to the constructor of `Polygon`. Compiling and failing because the constructor has yet to be implemented, the driver is then guided to code up the class `Polygon` in the file `polygon.h` with just the required constructor. After succeeding, the driver writes the assertion, compiling and failing again because the getter method is not yet implemented. The driver is then guided to code up the getter, compiles and runs the test until it passes.

Our experience shows that the TDD style process seldom became stuck when the navigator or the driver did not know how to proceed; someone in the class would always come to the rescue. In any case, there's always the instructor, but the instructor should give hint to nudge the process into the right direction rather than give out an answer straight away.

The process can change hand every time a test is completed, ensuring a broad participation. Usually, a test is completed within a few minutes.

Interestingly, our surveys showed that the students would prefer more time allocated to student coding in class even though they stated demonstration by instructor was helpful. Figure 4 shows the response of a recent survey regarding in-class coding.

In doing pair programming or mob programming, it is important to source as many different student drivers as possible. To this end, we began with volunteering and then gradually widened into targeted invitations. We have observed that students who are better at coding tend to volunteer more frequently. Having drivers with good coding skill could set the proceeding at a faster pace, but at the risk of frustrating other students.

We have tried mob programming on a medium size class (20+ students) and large size class (60+ students) in two different physical instructional environments. While mob programming by novice programmers of this scale was

```
#include "../src/vector.h"
#include "../src/polygon.h"

TEST(PolygonTest, ConstructPolygon) {
  double v1[] = {0,0};                        // designing test data
  double v2[] = {3,0};
  double v3[] = {3,4};
  Vector a( v1, 2);
  Vector b( v2, 2);
  Vector c( v3, 2);
  Vector vertices[] = {a,b,c};
  Polygon p(vertices, 3);                     // determining constructor signature
  ASSERT_EQ(3, p.sides())                     // getter Polygon::sides() followed by assertion
  ASSERT_EQ(3,p.vertex(2).at(1));             // Vector(3, 0) is the second vertex
  ASSERT_EQ(0,p.vertex(2).at(2));
}
```

Fig. 3. listings of unit test

unusual, it seemed to work reasonably well in both cases. The medium size class seemed to go more smoothly than the large size class, though it is not yet clear to us whether class size played a role in affecting the result.

4. SUPPORTING AFTER-CLASS CODING WITH CONTINUOUS INTEGRATION

Students gained depth on language features, design, and agile practices by doing non-trivial homework assignments. Based on the code and tests available from in-class coding, the instructor and the teaching assistants worked together to prepare additional requirements to be met. In a nutshell, a program submitted was expected to pass a given set of unit tests. We distinguished two types of tests: those that were disclosed to the students and those that were not. The former guided the students to meet the minimum requirements; a submitted program that passed the disclosed tests received a passing grade. However, to get a better grade, a program needed to pass as many undisclosed tests as possible.

We designed a setup to handle homework submissions using the free repository gitlab[5]. Students were asked to register for an account and create their individual private projects on gitlab. They then shared the private git-generated SSH keys with the teaching assistants and set up web hooks from their gitlab accounts to the Jenkins[6] continuous integration server managed by the teaching assistants. The students submitted homework by committing to their individual projects, which triggered the Jenkins server to retrieve the most recent commit, compile and run all tests - including those undisclosed. The students checked the result of the build immediately after to decide whether to continue improving for better grades. Thus, effectively, the Jenkins server executed a remote build and provided instant feedback to the committer.

5. CONCLUSION

We have reported our experience in using problems of a sufficient complexity to carry out teaching language features, design, and some agile practices including iterative and incremental development, unit testing, test-driven development, pair/mob programming, and continuous integration in an object-oriented programming course. While this requires more preparation by the teaching staff, we have seen encouraging feedbacks from the students. The survey result of the fall 2014 OOP course offering of Computer Science and Software Engineering at Auburn

[5] `https://about.gitlab.com/gitlab-com/`
[6] `https://jenkins.io/`

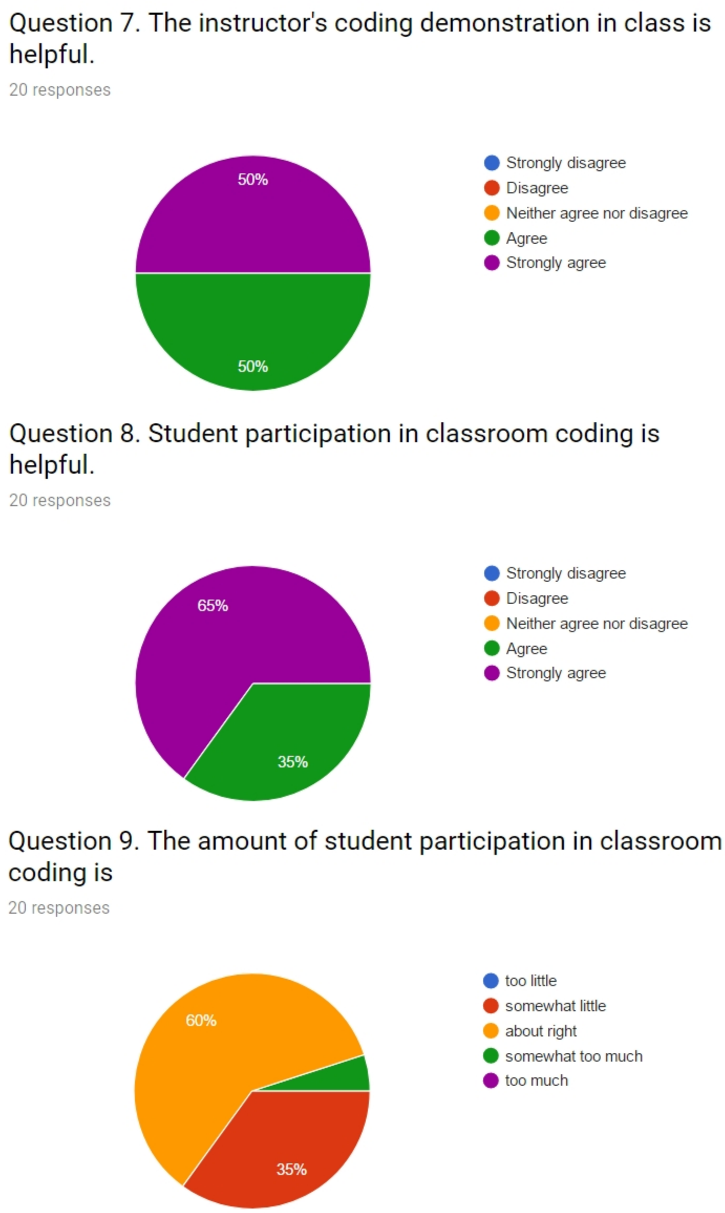


Fig. 4. Measurement of how helpful the students think of in-class coding in EECS OOP offering at Taipei Tech, spring 2016

University can be found in the appendix of reference [10]. Here, we have summarized below some of the interesting results of the spring 2016 survey taken by students in EECS program at Taipei Tech[7].

Regarding how confident the respondent was in handling the development of a program up to a thousand lines of code, the percentage grew from 35% before taking OOP to 80% after. The result was in agreement with

[7] `http://www.cc.ntut.edu.tw/~yccheng/oopsurvey/2016s.pdf`

our expectation since the homework assignments were built on the examples in class, which easily exceeded a thousand lines of code.

Regarding how helpful in-class coding was by instructor and by peers, the results showed that while the students strongly agreed that instructor demonstration was helpful (50%), they would prefer to see more coding done by the peers (65%); see Figure 4. We attributed the result to the students' willingness to participate in pair/mob programming.

Finally, regarding how challenging the OOP course was and how hard the respondent worked, 50% of the respondents thought the course was challenging and 90% said that they worked hard for it.

Acknowledgments

This research is supported in part by the grant MOST-105-2221-E-027-076 of the Ministry of Science and Technology of Taiwan and by Auburn University.

REFERENCES

The Joint Task Force on Computing Curricula. Computing curricula 2001 computer science. Technical report, New York, NY, USA, 2001.

Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, June 2003.

Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

Woody Zuill and Kevin Meadows. *Mob Programming A Whole Team Approach*. leanpub.com, 2016.

Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

George Polya. *How to solve it: A new aspect of mathematical method*. Princeton University Press, 1957.

Y. C. Cheng. Applying how to solve it in teaching object-oriented programming and engineering practices, 2014. Accessed: 2017-05-26.

Y. C. Cheng and K. H. Chang. How to solve it: Patterns for learning and teaching object-oriented programming and engineering practices, 2015. Accessed: 2017-05-26.

Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Pearson education, 2001.

Received December 2019; revised February 2020; accepted April 2020

AsianPLoP 2020, MARCH 4–6, Taipei, Taiwan. HILLSIDE 978-1-941652-03-9