Applying *How To Solve It* in Teaching Object-Oriented Programming and Engineering Practices

Yu Chin Cheng
Department of Computer Science and Information Engineering
Taipei Tech, Taiwan
yccheng@csie.ntut.edu.tw

Abstract. One way to teach undergraduate students object-oriented programming (OOP) is to develop programs for solving problems that are reasonably complex and which require the use of engineering practices such as testing, refactoring, error handling, version control, iterative and incremental development, and so on. As a result, side-by-side coverage of OOP language features and engineering practices is necessary. Since the time available for classroom teaching is limited, several conflicting forces are at play in such a context. This article presents, in a pattern format, an application of George Polya's *How To Solve It* to resolve the conflicting forces in developing and using complex and long-running programming examples for use in classroom teaching and learning.

Keywords: object-oriented programming, engineering practices, heuristics for problem solving, example design, classroom teaching and learning

1. Introduction

With its features to support object abstraction, collaboration, inheritance and polymorphism, and so on, object-oriented programming (OOP) is good for developing complex programs for use in the real world. Thus, one way to teach students OOP is to develop programs for solving problems that are reasonably complex [1]. On the other hand, developing complex a program requires the use of engineering practices such as testing, refactoring, error handling, version control, iterative and incremental development, and so on [2]. As a result, it seems that learning of OOP and learning of engineering practices should take place side by side. To the students, learning OOP for the first time in a course offering through complex examples can be overwhelming, let along adding the engineering practices. Clearly, there are conflicting forces at play in such a setting if we want to use programs of a reasonable complexity in teaching OOP and engineering practices.

This article reports my experience of applying the four steps of George Polya's *How To Solve It* [3] – *understanding the problem, devising a plan, carrying out the plan,*

and *looking back* – to resolve the conflicting forces in teaching OOP and engineering practices to sophomores who know C programming and getting on to learn OOP with C++ for the first time. *How To Solve It* is a classic on heuristics of solving mathematical problems and is filled with useful suggestions and questions for guiding problem solving. For a *problem to find*, the suggestions and questions for *Understanding the problem* include *What is the unknown? What are the data? What is the condition?* For *Devising a plan*, the questions include *Have you seen it before? Do you know a related problem? Look at the unknown!* And so on. Each of these is explained and illustrated in an article in the third part of the book called a "Short Dictionary of Heuristic". Indeed, each of these individual suggestions and questions can be seen as a pattern in itself. Since Polya's *How To Solve It* depends on the application of these suggestions and questions, it can be seen as a pattern language.

The present article reports my experience in applying *How To Solve It* in a pattern format. Associated with it are some questions and suggestions which I have found to be useful in developing and using complex and long-running programming examples for use in classroom teaching and learning.

2. How to Solve It

Context: Undergraduate students with first experience of programming (e.g., those who have programmed in a procedure language like C) move on to learn object-oriented programming (e.g., with C++) in a course offering. The students have the capability to write programs with sizes up to a couple hundred lines of code. They also have a very limited knowledge of engineering practices that are generally useful for developing software.

Problem: How do we teach object-oriented programming and engineering practices using reasonably complex examples?

Forces:

- Object-orientation is best learned with programs of a reasonable complexity.
- Engineering practices are required to develop programs with complexity.
- A typical course offering in object-oriented programming has a limited amount of time for lecturing in class and practicing outside class.
- Detailed coverage of language features can be time-consuming.

Solution: Prepare long running examples for use in class and guide the students to

solving the programming problem *incrementally and iteratively* in four steps: (1) *understanding the problem*, (2) *devising a plan*, (3) *carrying out the plan*, and (4) *looking back*.

(1) Understanding the problem: Spell out the *principal parts*: *What is required? What is given? What are the constraints?*
(2) Devising a plan: Having understood the problem, list the tasks needed to solve the problem on hand. Include in these tasks ones of learning language features and engineering practices.
(3) Carrying out the plan: Do the tasks by completing one task at a time and verifying that it is done correctly. Complete the learning tasks just before the working knowledge is needed.
(4) Looking back: Ask whether the problem is solved with the program written so far. Run the program to look for improvements that can benefit the user. Inspect the program to look for improvements in code and tests.

Before the class, prepare a problem of a reasonable complexity but one that makes sense to the students. Embed learning of language features and engineering practices in the process of solving the problem. Break the problem down into sub-problems so that a sub-problem – one that you intend to begin solving with – is within the students' current programming knowledge to solve (*understanding the problem*). Write down the reasons why you think this is a reasonable decomposition of the problem. One of the best ways to do this is to keep a journal of what you do, e.g., in a blog.

Begin the first iteration by breaking the selected sub-problem down into a number of tasks (*devising a plan*). List the tasks by starting with the required and working backward, or by starting with the given data and work forward, or both. Once you obtain a list of tasks, work through tasks one at a time; write up code that only makes use of features the students already know, and write it so that the completion of a task can be confirmed. Use a version control program (e.g., git [9]). Check in each time a task is completed. Obtain a working program at the end of completing all tasks (*carrying out the plan*). Run the program and consider what can be improved. Write down the improvement items. Identify the new language features and engineering practices that can be applied to improve the program in a convincing way (*looking back*).

Continue on with the second iteration. Consider to go on to the next sub-problem or to work on one or more improvement items. *Pick the latter when the new features or engineering practices required to make the improvements are now within grasp by*

the students. Prepare items of discussion with the students why this is a good choice (*understanding the problem*). List the tasks that are required to complete the improvement. The tasks can explicitly include *tasks of learning,* each one of which is listed just before the task that requires it (*devising a plan*). In completing a task of learning, make use of small and simple examples that focus on what you want the student to learn. It is highly recommended that you look for such examples on the Internet as many good examples can be readily found. Get the small examples to compile and run and get them over with quickly. Then apply the just-learned feature or engineering practice immediately to solve the next task that required it (*carrying out the plan*). End the iteration with looking back.

Alternatively, if a new sub-problem is chosen, consider what impact it brings to the program on hand (*understanding the problem*). If a new feature is added, how does it affect the existing features? What new program constructs must be added? What modifications to the existing constructs are needed? List the tasks (including tasks of learning where appropriate) according to the analysis (*devising a plan*). At the completion of all tasks, ensure that both the new features as well as the existing features are functioning as intended so that you again have a working program (*carrying out the plan*). Again, do the looking back.

Continue on with the next iterations until all sub-problems are done and no more improvement items remain. When finished, you have a good example you can use in class.

Just before a class, review the previously prepared materials you intend to cover. The journal you keep and the version control program will be of great help here. Make sure that the classroom is adequately equipped for you to write programs with the students in class. The following setup is suggested:

- A networked computer or notebook with which you and your students write the program and search for information about language rules and examples on the internet. The computer should have the required software already installed and properly configured, including a compiler, a build program, an IDE, a version control program, a unit testing framework, and a browser;
- An LCD projector to display the program and other class materials. Check the LCD projector's compatibility with the networked computer; and
- An extra pair of wireless keyboard and mouse, which is passed to the student who accepts your invitation to participate coding in class. Make sure that the wireless transmission range is long enough to cover the entire classroom.

In general, the networked computer should be *the only computer in use* in class. By allowing students to use their own computers in class, you risk losing their attentions and participations.

In the class, retrace what you went through in preparation, but this time strategically engage the students by asking questions and soliciting answers. Help the students understand the problem as a whole. Ask the following questions and soliciting answers: *What is required? What is given? What are the constraints?* Break the problem down into sub-problems with help from the students. Then focus on the sub-problem on hand (*understanding the problem*). Invite the students to break the sub-problem down into a number of tasks. Write the tasks down (*devising a plan*). Work with the students to write up the program on the computer, one task at a time. For learning tasks, retrieve the examples you previously identify during the preparation. Copy the code into the IDE, compile, and run. Check that the result is as expected. Then go through the program, focusing the features highlighted in the example. Work to obtain a working program when all tasks are completed (*carrying out the plan*). Run the program and ask the students: *Are we done solving the sub-problem? What can we do to improve the program for the user? What can we do to improve the program as a programmer?* Write the improvement items down (*looking back*).

New iterations are entered until all sub-problems and the selected improvement items you want to cover are solved.

Outside the class, build homework assignments around the problem you solved with the students in class. Ensure that the students put the learned features/practices in use. Provide reference implementation right after the students turn in the homework. Walk through the reference implementation with the students. If the running example takes several weeks to cover in class, the students should be given more than one assignments to grow the program in this period. The reference implementation distributed to the students after a due date will help the students getting back on track even if they got stuck on a previous assignment.

Resulting context: The long running examples cannot possibly be as complex as real world applications. Even so, the programs are complex in the sense that they are grown in a few weeks within the coursework. Appropriate language features and engineering practices are introduced just before they are applied. The learning takes place in a focused manner because the students are familiar with the problem on hand. In contrast, most examples seen in most OOP textbooks tend to be short and

feature-specific; learning different features often imply juggling around with a number of different and unrelated examples.

As the iterations go by, the students witness the program size grows in several measures: lines of code, number of classes and functions, number of files, and number of unit tests. In other words, they witness how the program grows in complexity. Having participated in writing the program, the students are less intimidated by the complexity of the program than if they were to study the completed program by themselves. Whether this is so can be easily checked by their ability to do the homework assignments, which are based on the example in class. Thus, the instructor should always check how the students perform in homework.

The four steps of *How To Solve It*, having been practiced in class and in homework assignments, will have a better opportunity to become a working habit of the students, one that can help them in future development work in which learning occurs spontaneously in a similar manner to what we describe here.

Known use: The four steps were suggested by the great mathematician George Polya for solving mathematical problems [3]. It has been applied in computing as well, data structures and algorithm [4]. Although not emphasizing the process aspect, it has been applied in analyzing and structuring software development problems [5].

3. How To Solve It in action

At CSIE department of Taipei Tech, OOP is a required course for sophomores who have completed six credits of programing in their freshman year. Entering OOP, registrants are capable of writing programs up to a couple hundred lines of code in the C language. After completing OOP, the students will go on to take OOP Lab to work in a game in two-person teams to create game programs in C++ of a size up to 3,000 lines of code [6].

Thus, being the course to bridge freshman programming and OOP Lab, it is expected the OOP course will enable the students to handle developing C++ programs meeting the following descriptions: code size up to a 1,000 lines of code with unit tests; number of classes between 5 to 10; making use of object abstraction, composition, aggregation, virtual functions, abstract classes, and templates; making use of modules from the standard library C++ including stream I/O, containers, iterators, and algorithms; simple but effective use of exception and exception handling; organizing the program into files for the ease of unit testing and efficiency of builds; writing acceptance tests for their program; and utilizing version control.

The following description is an account of the application of *How To Solve It* in the

Fall, 2013 course offering of OOP.

Before the class. The instructor prepared two long running examples for class use. The first example, a problem involving vector and matrix computations, aimed to make the transition from C programming to C++ programming with two emphases: encapsulating and abstracting with objects and the basic practices as summarized in Table 1. The second problem, a problem of managing the registrations and course elections in a simplified college setting, moves on to object collaborations, inheritance, polymorphism, and the use STL containers, iterators, and algorithms (Table 2). In both examples, the instructor applied the problem frame approach to decompose the problem into sub-problems [5].

As can be seen from Table 1 and Table 2, a mix of selected engineering practices and OO principles are covered. While tempting, a comprehensive coverage of engineering practices and OO principles is not possible due to time limit. Adopting HTSI, it is the responsibility of the instructor to design the long-running examples to cover the practices and principles as appropriate. In this case, knowing that the students have a very limited programming experience entering the course, I selectively cover just a small number of practices and principles for managing dependencies, especially from the perspectives of keeping unit tests simple to write and keeping unit testing modules independent. I have also covered, late in the course, the open-close principle [11].

In what follows, I will cover the first example in some detail. The first example is a problem involving vector and matrix computations, which is chosen because the students have completed a required course on linear algebra in the freshman year and know the computations (*example that makes sense*). The computation involves computing inner product of two vectors, applying a linear transformation to a vector, computing a composite linear transformation by matrix multiplications. The program also involves console and file I/O processing. The program was completed in 6 iterations (approximately six weeks). The instructor breaks the problem down into three sub-problems: computing inner product, computing linear transformation and computing composite linear transformation.

Table 1. Example 1: vector and matrix computations

Iteration	Language and programming	Engineering practices
1	Reviewing of C programming; C++ console output; operator overloading	How To Solve It (HTSI); devising a plan – listing tasks; looking back – as a user; working program at the end
2	Function overloading; use of	Understanding the problem –

	return code; runtime memory model of programs	prioritizing; looking back – as a programmer
3	Encapsulating with object – data member, member function, private; public, constructor, and desctructor; passing parameters by reference;	Understanding the problem – analyzing the improvement items
4	Separating class definition (.h) from class implementation (.cpp)	Unit testing with CppUnitLite [7]; modularity – separating main program from tests; modularity – separating build targets;
5	Composing C functions that use objects; C++ file I/O; string;	Test-driven development; managing dependency for unit testing; looking back – as user for robustness; looking back – as programmer for exposing bugs
6	Replacing return code with exceptions; exception handling – fail fast; exception handling – involving the user;	Unit testing exception handling;

Table 2. Example 2: college registration and course election

Iteration	Language and programming	Working knowledge
1	Collaboration; composition with value object; aggregation with pointer and reference;	Version control with git
2	Templates; STL: containers and iterators – vector; Algorithm: sort	Stretching the unit testing framework – Integration testing
3	Adding more classes and more functionality	responsibility assignment ; CRC cards;
4	Managing dependencies – boundary objects	Acceptance testing with Robot Framework [10];
5	Inheritance; virtual method and polymorphism; Open-Close Principle [11]	
6	Managing dependency –	Refactoring tests by namespace

	namespace	

In the class. The first iteration quickly produces a *working program*, which prompts the user for two vectors and calculates their inner product when calculable. The simplicity of the problem makes room for a concise treatment of "How to solve it". Building on the students' previously acquired knowledge of C programming, the first three steps of HTSI are quickly done in class; much emphasis is put on the step *looking back*, in this case as a *user*. The instructor invites a volunteer to start up the console mode program and interact with it. After a couple of quick runs that ended in success, the volunteer is asked to find fault with the program, which he did with little trouble by deliberately typing in two vectors of different dimensions. The program responded by printing "Dimension error!" and then terminated. The students were asked whether this is acceptable if he/she is the user, and if not, what improvement is necessary. After some discussion the students generally came up with an improvement item: "the program should keep on running rather than aborting." The instructor quickly registered the improvement item. The instructor then prompted further: what else can cause *the program to crash*? Again, after a few brief exchanges, the volunteer typed in a vector that is syntactically incorrect (e.g., (1,2] or (0,a)). The program crashed accordingly. Another improvement item was recorded. *Looking back* was done in a matter of minutes.

Observation. Other concerns aside, the students are given confidence that they are able to solve the problem. Nothing new is taught just yet but the students are made aware that obvious improvements are needed.

Iteration 2. Starting the second iteration, in the *understanding the problem* step, the instructor has three options: he can choose to work on linear transformation, composite transformation, or to do the improvements first. The instructor solicited opinions from the students, maneuvering for the students to reach an agreement to do the improvements first, e.g., by suggesting if the program were to be delivered early, do they prefer providing a program with three fragile features or one with a single robust feature? The instructor also asked the students to speculate whether the improvement problem will become more difficult if postponed. Iteration 2 is then settled on improvement.

While completing the next two steps, the instructor covered the topic of overloading a function and provides two versions of the function "innerProduct": one that makes forced exit upon dimension error and the other that returns false to let the caller (in this case the main program) to continue.

In the looking back step the instructor again had a working program. This time he asked a volunteer to make sure that the improvement is accomplished. He then guided the students to look at the code, asking questions and soliciting answers while he did so. *Is the program easy to read? Is it easy to add new features, since two more will be added later? Does the program contain code that can easily lead to bugs?* And so on. Such questions lead to the identification of several improvement items, which serve to motivate learning new C++ features. For instance, the students quickly identified a culprit:

```
void outputVector(double v[], int dim);
```

which takes two parameters to specify the vector to be printed. Suppose vector u is of dimension 2 and vector v of 3. While outputting the vectors, the programmer can mistakenly type

```
outputVector(u,3);
outputVector(v,2);
```

The programmer may not easily discover the mistakes because compiling it incurs no errors and no warnings, and running it demands his attention to the output spat out on the console, which can easily escape the due attention. The fact is recorded: "A vector should *remember* its own dimension."

Observation. Looking back from the programmer point of view identifies and motivates improvements in the design and code of the program. It also serves as a good cause for motivating learning new features.

Iteration 3. Favoring improvements, the instructor creates an opportunity to cover class and object for the first time. In *understanding the problem*, the instructor analyzed the problem; the associations among the concepts (e.g., dimension, components, etc.) centering around representing a vector were identified:

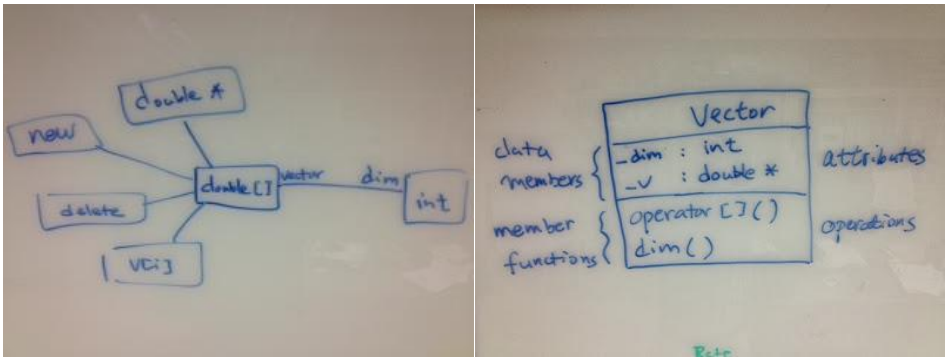

Figure 1. Making a case for encapsulating with object

The diagram (left of Figure 1), which the instructor drew with help from the class, shows that a vector is an array of doubles; “double []” is equivalent to “double *”; a vector is allocated with the “new” operator and de-allocated with the “delete” operator; and a vector is associated with an “int” for its dimension; etc.

Having now understood the problem, the instructor announces the good news that the associations of data and operations around vector are achieved with a properly defined C++ class and produced the diagram (right of Figure 1). The opportunity, which he worked with the students to create, put him in a good position: for he can now determines the minimum amount of features of the class to cover. Accordingly, in *devising a plan*, the problem is broken down into the tasks of defining Vector and making modification to functions to use it. A learning task is added before the actual task (*devising a plan*).

In drawing the diagrams, the related UML notations were introduced.

In *carrying out the plan*, the instructor quickly covered defining and implementing a class using a small example the instructor located on the Internet. The features were put in use immediately, and accordingly the data members, member functions like constructor, destructor, accessing a component, and obtaining dimension, are created. In so doing, the instructor checked out the right version of the program which he prepared before class. He strategically wrote up some code in class, compiled and executed the program, soliciting help from the students while so doing.

In *looking back*, the instructor pointed out to students that main program contains “tests” for printing out test results to the console, which has the drawbacks of being time consuming (the main program edited to switch between testing and computing inner product for the user), output easily ignored, and so on. This fact was noted.

Observation. Many start object-oriented programming by declaring “Everything is object”. While the adage are well-known to people already know OOP, to students beginning to learn OOP, it brings confusion: Is integer 3 an object? What are its

operations? And so on. These questions are not easy to answer, and answering them at the first opportunity of learning object and class carries the risk of getting into philosophical debate that tends to confuse the students. In contrast, by setting the goal to improve the existing program, the extent to which vector should be objectified is clear and avoids unnecessary questions to be brought forward at this early stage.

Iteration 4. The instructor oriented the students to work on the problem of improving the way that program behavior is checked with unit tests. A unit testing framework, in this case CppUnitLite [7], was used to separate the test code from the main function and different build targets were defined in makefile of the build tool "make". The fact that failed tests displayed the line failure occurs convinced the students of the merits of unit tests.

Observation. Students saw for the first time the neat output produced by unit tests, which contrasts sharply with the old way of dumping test result to the console. Also, the use of "make" allows main program and tests to be built separately.

Iteration 5. The instructor decided for the class that linear transformation will be the next sub-problem tackled. The instructor analyzed whether to add linear transformation as a member function of the class Matrix or as a C function. The latter was preferred because it avoids adding dependency from matrix to vector, thus making the unit tests of matrix separate and self-contained (*understanding the problem*). Tasks were listed (*devising a plan*). The instructor took this opportunity to demonstrate how unit tests for Matrix and the C function for linear transformation should be written – in separated files – before the function and the class were implemented (*carrying out the plan*). The students were shown the benefit of modularity: the class Vector and it unit tests were unaffected and did not need to be recompiled and rerun. Now that the program has more functions, the instructor invited the students to review it again both as a user and as a programmer. It was found that the code for error handling seemed ad hoc and depended on the use of return code, which forced functions to have more parameters. In the case of the member function that overloaded operator [], return code was not possible at all. These observations were written down (*looking back*).

Iteration 6. The instructor suggested tackling the error handling problem. Using the observations made in the previous round, he further asked the students what one should do when the error involves the user (e.g., input a dimension incorrectly) or

when the error reflects a mistake made by the programmer (e.g., accessing an out-of-bound component of a vector). For the former, the strategy is to *involve the user: program should tolerate the user's operation errors*; and for the latter to *fail fast* [8] (*understanding the problem*). He listed tasks accordingly, including two for learning exception and exception handling (*devising a plan*). The instructor then worked on the unit tests first, focusing testing for exception this time. He suggested throwing an exception as a string that describes the cause of exception. With exception, many functions are simplified both at the definition and at the implementation. The user is given an opportunity to fix the error made and to retry the operation.

At the conclusion of this example, which took place around the eighth week of an eighteen-week semester, the students had their first encounter with unit testing, test-driven development, modularity with make for separated files, encapsulation with object, managing dependency for unit testing, simple exception handling, reviewing as a programmer, and reviewing as a user. The learning is reinforced by a healthy volume of homework built around the example covered in class.

4. Conclusion

This paper describes an experience of applying *How To Solve It* in teaching object-oriented programming and engineering practices to undergraduate students with a background of C programming. A number of long running examples have been developed and used in class. The use of such examples has been instrumental in reaching the goal set for the object-oriented programming course the author taught in the past five years. On the other hand, while the students seemed largely positive about the HTSI-style of class teaching-learning, more long running examples should be developed and used in classroom by applying HTSI. Also, more quantitative and qualitative assessments need be performed to validate the claimed benefits of applying HTSI.

Acknowledgement. This research is supported in part by the National Science Council of Taiwan under grant contract 102-2218-E-027-005. The author wishes to thank Dr. Foutse Khomh for shepherding this pattern.

References

[1] Chris Nevison and Barbara Wells. 2003. Teaching objects early and design patterns in Java using case studies. SIGCSE Bull. 35, 3 (June 2003), 94-98.

[2] Kent Beck and Cynthia Andres. 2004. Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional.

[3] George Polya. 1957. How to solve it. Second ed. Princeton University Press.

[4] R. G. Dromey. 1982. How to Solve it by Computer. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[5] Michael Jackson. 2000. Problem Frames: Analyzing and Structuring Software Development Problems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[6] W.-K. Chen and Y. C. Cheng, “Teaching Object-Oriented Programming Lab with Computer Game Programming,” IEEE Transactions on Education, vol. 50, no. 3, pp. 197-203, August 2007.

[7] CppUnitLite, http://www.objectmentor.com/resources/bin/CppUnitLite.zip. Accessed 6 Jan 2014.

[8] Shore, J., "Fail fast," Software, IEEE , vol.21, no.5, pp.21,25, Sept.-Oct. 2004

[9] git, http://git-scm.com/downloads. Accessed 6 Jan 2014.

[10] Robot Framework, https://code.google.com/p/robotframework/. Accessed 6 Jan 2014.

[11] Robert Cecil Martin. 2003. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River, NJ, USA.