How To Solve It: Patterns for Learning and Teaching Object-Oriented Programming and Engineering Practices

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

Kai H. Chang
Department of Computer Science and Software Engineering
Auburn University, Auburn, Alabama
changka@auburn.edu

Abstract. *How To Solve It* is a previously published pattern for planning and teaching a course on object-oriented programming (OOP). It has been successfully applied in a number of OOP course offerings to undergraduate students of computer science and software engineering. In this paper, refined patterns to support the main pattern *How To Solve It* are presented. The patterns are presented with the intention that the learner is the reader, though it can readily be applied by an instructor in planning and teaching a course in OOP. Our most recent experience in applying these patterns in teaching OOP as a second course to students outside of computer science and software engineering is reported.

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

1. Introduction

Practical problems are complex in nature. When developing programs for solving practical problems, the developer combats the complexity by uncovering the structure of the practical problem on hand. He then breaks it down into smaller sub-problems, develops solution for each, and combines the solutions incrementally. The solution scenario so described is at the heart of many popular software development methods including the unified process [Lar2012], agile methods [Bec2004], and so on.

To prepare students to develop programs for practical problems, a typical curriculum of computer science (CS) and software engineering (SE) includes multiple

courses covering introduction to programming, data structures and algorithms, object-oriented programming (OOP), software engineering, and capstone project. However, a non-CS&SE curriculum typically offers no more than two courses, usually just introduction to computer programming and object-oriented programming. The former often covers procedure-based programming using a language such as C, BASIC, FORTRAN, Python, MATLAB, etc. The latter builds on the former by covering features of an OO language (e.g., C++, Java, Python, etc.) and object orientation concepts. OOP has been the prevailing paradigm for the second course since it is currently the most widely-practiced programming paradigm for developing practical applications in the real world.

Here is the question: *How do we prepare non-CS&SE students for their roles as programmers in developing practical applications in OOP as the second course?* While teaching OOP to CS&SE students is not an easy task [Ber2000][Che2014], teaching OOP to non-CS&SE students is even more challenging due to the limitation imposed by curriculum. Furthermore, it is not enough to just cover language features and OO concepts. Methodologies and engineering practices for iterative and incremental development are also needed. However, covering the latter in a typical way is out of the question, or at least will be extremely difficult due to the curriculum limitations. Thus, it is necessary to narrow down to essential aspects that can be covered in a limited time, e.g., one semester.

This paper presents five patterns that can be used to plan and teach OOP as a second course for non-CS&SE majors under the umbrella (or skeletal) pattern called *How To Solve It* [Che2014]. Details of pattern *How To Solve It* can be found in Appendix A. The five patterns of this paper are written in response to the numerous feedbacks with respect to [Che2014] from the shepherd and the participants of Writers' Workshop – Group A. In a nutshell, the main point is that *How To Solve It* should be made a pattern language rather than a single pattern. Thus, the five patterns of this paper are intended to cover the four phases of activities in *How To Solve It*: *understanding the problem*, *devising a plan*, *carrying out the plan*, and *looking back*. Also, pattern *selecting and sequencing* is applied throughout the four phases since there is not enough time to cover all possible topics of OOP and engineering practices in a one-semester course. The patterns are presented in a form easy for the learners' consumption. That is, the students should be able to take the patterns and practice iterative and incremental OOP to develop practical applications. Applying the patterns in course instruction, the instructor should first try to develop

long running examples with these patterns. This will contribute to the selection and sequencing of the materials to be covered.

2. The Patterns

The patterns are presented in the following six-section format: the name (with proper capitalization for ease of identification), the context, the problem, the forces, the solution, and the consequences. The patterns are presented in the order they are applied.

2.1 Understanding the Problem

Context: You are given a problem whose inputs, outputs, and constraints are given in the problem statement. You do not yet know what will be involved in the development tasks ahead. You have decided to proceed with iterative and incremental development described in *How To Solve It* [Che2014].

Problem: Although the problem's inputs, outputs and constraints are given, you don't know how they are related to each other. You know it would be difficult to solve the problem in its entirety by considering all inputs, outputs and constraints simultaneously, thus you have decided to proceed with *How To Solve It*. What should you do to benefit from this decision?

Forces:

- A problem can be too large or too complex to solve in one round.
- The problem might have hidden aspects you don't yet know.
- You don't want to spend too much time analyzing the problem.
- You want to have a good grasp of the problem as a whole throughout the development.
- You want to prioritize on the most important problems.

Solution: *Break the original problem down into a number of smaller (sub-)problems if necessary*. This is similar to the heuristic of *Decomposing and Recombining* for solving mathematical problems [Pol1957]. When there are many required outputs, consider them separately. Decide what inputs and constraints are applicable for each separated output. Each separated subset of outputs, inputs and constraints points to a sub-problem within the original problem. *Write down the problem statement for each sub-problem*. There are many ways to do this: use cases [Lar2012]; user stories

[Coh2004] [Pat2008]; framing the problem against known problem types [Jac2001]; and so on.

Identify the interactions between individual problems. If a problem closely interacts with another problem, combine them. If the resulting problem is too big, refine them into equivalent problems such that the strength of interaction is reduced [Ale1964].

Put all problems in the problem backlog. Only problems in the backlog will be solved, if they are solved at all. *Prioritize the problems* with pattern *Selecting and Sequencing.* For the current round, select the problem with the top priority. *Keep the problem backlog up to date* (*Looking Back*).

Problem decomposition in this pattern is related to but different from *divide-and-conquer,* a technique that divides a large problem into two or more sub-problems that are of the same type as the original problem, but smaller [Cor2001]. In this pattern, the sub-problems are usually of different types; see [Jac2001] for more details.

Consequences:

- The problem to be solved is better understood.
- A collection of problems, each of which smaller than the original problem, is obtained.
- The original problem is solved piecemeal.
- Any problem to be solved goes through the problem backlog.
- A problem of high priority gets solved before a problem of low priority.

2.2 *Selecting and Sequencing*

Context: You are applying *How To Solve It* by going through the four phases of *Understanding the Problem, Devising a Plan, Carrying Out the Plan,* and *Looking Back* iteratively.

Problem: While so doing, there are often multiple items that attract your attention. You could have more than one problem in the problem backlog and more than one task in the plan. How do you decide which items to focus on next?

Forces:

- You want to pick an item to work on as you please.
- Since some of the items may be dependent upon each other, there is a logical ordering that exists among these items.

- Some items may not need to be done at all.

Solution: *Prioritize among the items***.** The dependencies among the items have been identified in *Understanding the Problem* (if the item is a problem) and *Devising a Plan* (if the item is a task). Logically, you should solve the problems by progressing from the least dependent to the most dependent. But such a sequence may not be what you want. In agile development, the customer gets to say which problems (i.e., user stories) have the top priorities and what the customer picks may not be the most independent.

In our current context of learning and teaching OOP and engineering practices, you may also need to go out of the logical ordering. Pedagogically, if solving an independent problem seems less important than solving other more dependent problems, you could choose to solve the more dependent problems first. But this comes with a cost: the dependent problems can be solved only if we pretend that the problems it depends on are solved. Usually, this involves writing code to separate the dependencies.

Some criteria are used while prioritizing the items. In the example of Section 3, the instructor selects the top priority problem by considering what to cover next among three categories of topics: methodology and engineering practices, language and libraries, and object orientation. This is equivalent to selecting and sequencing materials in learning technology [Che2009].

Consequences:

- Some overhead may be required if you go out of the logical ordering of the items.
- You are able to spend your time on the most important items.
- An item of high priority, when implemented, can impose additional constraints to the items yet to be implemented.
- You must have a set of well-defined criteria for prioritizing the items. Lacking such leads to an unjustified preference to one item over the other items.

2.3 *Devising a Plan*

Context: A problem with top priority has been selected for solution from the problem backlog. The problem statement has been written, and its inputs, outputs, and constraints are specified.

Problem: Having understood how inputs, outputs, and constraints are related in the selected problem, you are in a good position to start the implementation work for solving the problem. However, you still need to consider other aspects in addition to the implementation work, for example, what engineering practices to use and when to perform?

Forces:

- You want to proceed with the development work in small steps.
- You don't want to spend too much time analyzing the problem.
- You want to ensure that working solution is obtained after solving the problem.
- You want to avoid reworking the problem as much as possible when you solve another problem in a latter round.

Solution: Break the problem down into a number of tasks. Each task should be a *unit of work* you are comfortable of handling and can be completed in a time frame ranging from a few minutes to a couple of hours. *The tasks should be as independent as can be*. Distinguish tasks of four different categories:

- Implementation: there will always be implementation tasks because you have to write code to solve the problem. *Distinguish normal cases from exceptional cases.*
- Integration: you want to make sure you have a working program at the end of a round. Thus, *include an integration task for the problem you are solving*. The integration task can be a number of acceptance tests prepared according to the problem's inputs, outputs and constraints.
- Learning: before you use something you have not yet mastered, you must first learn it. Solving a problem gives you a clear context for learning. You will know how much learning is enough (*Carrying Out the Plan*), but first you need to know what to learn (*Selecting and Sequencing*). Search for online resources (e.g., tutorials, code examples, etc.) to help you plan learning.
- Rework: rework is an unavoidable part of iterative and incremental development. You might have begun with a strategy to make the program work. Later, however, when you learn an alternative way that seems to better support the subsequent development, you may change to the alternative way without changing the working program's external behaviors. Or, you might have discovered a bug in your program and decided to fix it. Rework tasks are mostly associated with improvement items identified in

Looking Back.

Organize the list of tasks as follows. First, *work out a list consisting of implementation and integration tasks* that constitute the essence in solving the problem (*Selecting and Sequencing*). *Refine the list with additional tasks*. For each task, consider what makes it difficult, if at all. If it involves something you don't know, add a learning task, e.g., learning a unit testing framework, learning to use a STL function, and so on. Or, it can be that the existing code makes performing the task cumbersome. In this case, add a rework task to fix the existing code, e.g., changing design, reorganizing project structures, and so on.

Consequences:

- The problem selected for solution is understood from the implementation point of view.
- Units of work are created and a plan of solution is obtained.
- Learning and rework are planned to take place alongside implementation and integration.
- Selected engineering practices are factored in as part of the implementation work.

2.4 *Carrying Out the Plan*

Context: You have obtained a list of tasks that must be performed for solving a problem. You are ready to spend your time learning, coding, testing, refactoring, and integrating. This is where you spend most of your time in a development round.

Problem: Except for sequencing of tasks identified in *Devising a Plan*, the tasks are as independent as can be. Therefore, you have a lot of freedom. How should you go about performing the tasks?

Forces:

- You want to finish the tasks in a timely manner.
- You are in the middle of performing a task but some other tasks seem more appealing to you.
- You have yet to familiarize yourself with the language features, APIs, or certain engineering practices that are needed for solving the problem.
- You want to make sure that implementation does not leave a lot of technical debts.

Solution: *Perform one task at a time*. Since the tasks are as independent as can be, you are free to perform them in an order that works best for you. Even so, there may be some good ordering rules you should follow (*Selecting and Sequencing*).

For a task of implementation, follow the steps: *write a unit test that fails* and *then write code to make it pass* [Bec2003]. By writing a unit test that invokes a function, you are deciding the name of the function, the arguments it takes, and its return type. This allows you to concentrate on implementing the body of the function later.

If learning is required for an implementation task, learn *just before* implementation. *Learn with unit tests*. For example, learn to use the standard template library (STL) sorting function by calling it and checking the result within unit tests. *Budget learning by writing just enough unit tests* for the features you need. For example, assume that you use a class for the first time and you need a constructor. Do you need to learn all the rules for declaring, defining and using a constructor? No! Just focus on the ones you need by writing a unit test for exercising the specific constructor. If you need other constructors, you can always identify them during *Looking Back*.

For an implementation, integration, and rework task, make sure *that all unit tests pass before you complete it.* Check in the code to the repository immediately afterward.

Consequences:

- An individual task is completed in a small amount of time. Signs of progress [Pol1097] are clear as you make your way through the tasks.
- The lead time from learning to coding is kept short by placing the implementation task immediately next to the learning task. Thus, the learning takes place just in time and with a clear objective.
- Learning is captured with unit tests. Later, when you need to recall what you have learned, you can just look at the unit tests.
- Budgeting learning with unit testing is an instance of applying spiral learning [Ber2000].
- Wastes associated with over-learning are avoided.
- Unit testing precedes implementation to drive design.

2.5 *Looking Back*

Context: You have written code and tests as required by the tasks that solve the problem for the current round. Code compiles clean; all tests pass; and the program

seems to run as intended. In short, the problem appears to have been solved.

Problem: Having solved the problem of the current round, you have made progress in solving the original problem. What are the new possibilities that open up given this progress? Further, has the problem of the current round been solved without hidden issues?

Forces:

- You want to move on to the next problem.
- You have a feeling that something in the solution is not good enough.
- You have just spent some length of time learning and writing code. You want to reaffirm what you have achieved.
- You want to be aware of the new possibilities open up as a result of solving problem of the current round.

Solution: *Inspect the working program as a programmer/reviewer*. Ask the following questions: Is the code easy to read? Is there code duplication? Are the tests easy to find and read? Is the code easy to modify? And so on.

Run the program as a user and explore. Does the program abort easily? Does it crash easily? Is it easy to use? List any finding as an *improvement problem*.

Do a retrospective/review on the current adoption of engineering practices and identify places where growth is possible. Are you writing unit tests? Are you writing unit tests before implementation code? Is your code version controlled? Again, list any finding as an *improvement problem*.

Re-examine the problem as a whole, taking into consideration the problems that have been solved so far. In iterative and incremental development, it is important to stay vigilant on the original problem to be solved as our understanding improves with some sub-problems solved. Add any new sub-problem that is omitted previously. Look for ways to reformulate the sub-problems in the backlog not yet solved.

Consequences:

- The current round ends with inspection. One of the things that upsets development tempo is a development round that drags on and on.
- Improvement items are identified and posed as problems and are put in the problem backlog to be weighed against other problems (*Understanding the Problem; Selecting and Sequencing*).
- Validate just-in-time learning. If the problem includes a learning task, the code written subsequently serves as evidence that the learning is completed

satisfactorily.

- Anticipate learning. An improvement problem may involve using new language features (e.g., encapsulation with objects), APIs (e.g., using a function or a container from the standard template library), changing design, changing project structure, and so on. The identification of improvement problem helps motivate learning new language features or engineering practices.
- Understand the program better. Through inspection, you become more familiar with the code you have written so far.

3. *How to Solve It* in action

This section presents the experience of running a course called, COMP3000 *OOP for Scientists and Engineers* at Auburn University in fall, 2014. Most of the enrollees majored in Wireless Engineering – Hardware, an undergraduate program offered by the Department of Electrical and Computer Engineering.

The instructor planned the course following *How To Solve It* principles. The problem was teaching OOP as a second programming course to non-CS&SE students who knew C but probably did not program in the previous year. The objectives were set as follows:

Objectives

1. Being able to handle the development of C++ programs up to 1K lines of code
2. Being able to exercise iterative and incremental development with *How To Solve It*
3. Being able to apply object-oriented programming principles
4. Being able to apply engineering practices including unit testing, acceptance testing, and version control.

The instructor designed the first long-running example problem and wrote the code. He decided to wait it out on designing the next examples, wanting to adapt the latter based on how the course progresses. In designing the first example, he followed the patterns in *How To Solve It*. Going through the process allowed him to sequence presentations (*Selecting and Sequencing*), in particular, in *Looking Back* to anticipate learning. It is worth mentioning that a lot of thoughts had to go into the *Understanding the Problem*. The instructor had to have a problem that would be unfolded well to anticipate the planned learning. As a result, the presentation sequence largely happened as the actual course progressed as will be described below.

The lectures were clearly tagged as one of three categories: *methodology and engineering practices*, *language and libraries*, and *object orientation* (*Selecting and*

Sequencing). The lectures were threaded with a three long running examples that involved computation with vectors, matrices, and convex polygons. Live coding was demonstrated by the instructor in class with student participations. Code written in class was made available to students. Homework assignments were built on the long running examples and made use of the code released. The instructor maintained a blog containing various articles related to the lectures [Che2014-2].

The lectures began with a one-hour introduction for *How To Solve It*. Then the first problem for live coding was given:

> Inner product, round 1.
>
> *Problem. Prompt the user to input two vectors. Compute the inner product (or dot product) of two vectors when it is defined. For example,*
>
> [1, 0] · [1, 1] = 1,
> [1, 1, 0] · [0, 1, 1] = 1, and
> [1,0] · [1,1,0] => undefined.
>
> *Prompt the user whether to continue or stop.*

Round 1. Proceeding with *How To Solve It*, the problem was divided into three sub-problems as shown in *Understanding the Problem.*

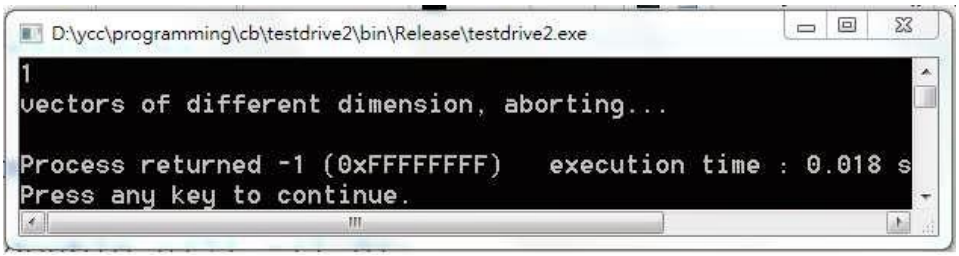
> Problem backlog
> P1. *Prompting the user*. Display a welcome message; invite user to input a vector, and get the vector.
> P2. *Computing the inner product*. Given two well-formed vectors, compute their inner product if defined. Flag an error otherwise.
> P3. *Displaying the output*. Display the output message that show the input vectors and the inner product.

The division into three sub-problems is straightforward and sought to make contact with the students experience in procedure-based programming. Arguing that problem P2 is the core of the problem (*Selecting and Sequencing*), the instructor went on to list the tasks to be done (*Devising a Plan*).

> T1. Write a function to compute inner product - normal case.
> T2. Handle the case where an inner product is not defined.
> T3. Write supporting program to exercise the function.

With help from the students, the instructor produced code for T1 and T3 (the happy path), followed by code for T2 (the exceptional path). T3 makes use of the main function (*Carrying Out the Plan*). A working program is produced. The instructor

then reviewed the code with the students and ran the program. A single file "main.cpp" contains all code. The function *main* has been used to run two tests and displays something that required some deciphering as shown below.



The observation motivated adding Problem P4 to the problem backlog:

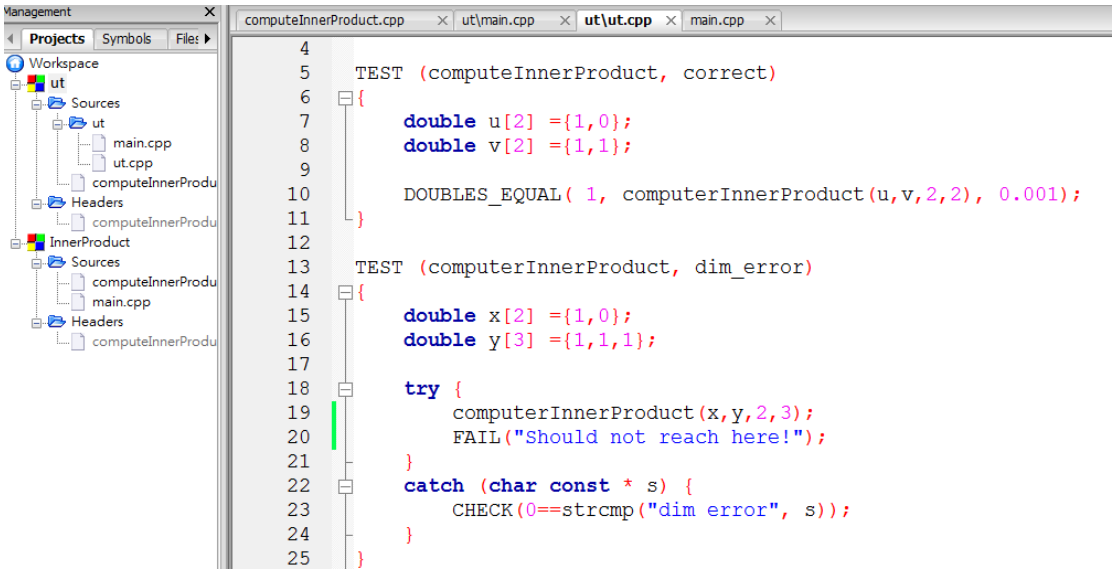
P1. *Prompting the user.*
~~P2. *Computing the inner product.*~~
P3. *Displaying the output.*
P4. *Test inner product computation in a separate project.*

Round 2. The instructor convinced the class to select P4, which gave him an opportunity to cover unit testing and exception handling (*Selecting and Sequencing*). Note that the two topics are usually covered much later in an OOP course, which means that the students don't usually get enough unit testing and designing and coding exception handling practice. The following tasks are listed:

T1. Prepare the CppUnitLite library.
T2. Set up a test project and practice writing unit tests.
T3. Change the function computeInnerProduct so that it can be tested in the test project.
T4. Relocate test cases from main to the test project.
T5. Learn exception, exception detection and exception handling.
T6. Replace forced program exit with exception handling.

The instructor proceeded to perform the tasks in the order listed. In task T3, code of the function computeInnerProduct is relocated into a header (.h) file and an implementation(.cpp) file. Task T6 took place after covering exception handling in T5 (*Selecting and Sequencing*).

The program structure grew from one file under one project to a three files under the production project "InnerProduct" and two files under the test project "ut"; see the left-hand side of figure below. The Code::Blocks IDE is used [Cod2013].



The two unit tests, which replaced the original tests in the main function, are shown on the right-hand side. C-style string has been used as the exception object thrown so that knowledge of C++ exception object was not required. The students were told that same program structure would be able to support all further development throughout the long running example. When executed, the console display looks like:

```
There were no test failures

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

In *Looking Back*, the instructor pointed out to the students the clutter-free screen showing success. He further played with the program by making the unit tests fail and showed the console output to convince the students the benefits of using a unit testing framework.

```
Failure: "expected 2.000000 but was: 1.000000" line 10 in C:\Users
\Downloads\InnerProductR2(2)\InnerProduct\ut\ut.cpp
There were 1 failures

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

The conclusion of round 3 marked the end of the second week.

Round 3. Problems P1 and P3 were combined into one problem and then solved (*Understanding the Problem*; *Selecting and Sequencing*). In *Looking Back,* in order to motivate the first introduction of object, the instructor invited the students to

inspect the code and identify code related to vector shown below.

```
 5  int main()
 6  {
 7      int dim1, dim2;
 8      double * vec1=0, * vec2=0;
 9      double ip;
10      bool cont = true;
11      while (cont)
12      {
13          try {
14              vec1 = getVectorFromUser(dim1);
15              vec2 = getVectorFromUser(dim2);
16              ip = computeInnerProduct(vec1,vec2,dim1,dim2);
17              std::cout << "Inner product is:" << ip << std::endl;
18          }
19          catch(char const * s) {
20              std::cout << s << std::endl;
21          }
22          delete [] vec1;
23          delete [] vec2;
24          cont = promptUserToContinue();
25      }
26      std::cout << "Terminating program..." << std::endl;
27      return 0;
28  }
29
```

The instructor showed the students how information of the vectors in the program scattered around and some effort was required to understand the code:

dimension of vectors (line 7)

pointer to array of doubles (line 8)

creation of vectors (lines 14 and 15; code not shown)

deletion of vectors (lines 22 and 23)

In particular, the programmer had to associate the dimension represented by variable dim1 with the vector represented by variable vec1, dim2 with vec2, and so on. As can be seen, the associations were made through naming conventions. Also, it was easy for lines 22 and 23 to be forgotten, resulting in *memory leak*. The inspection served to motivate learning a better abstraction through use of object. The instructor proposed to add the following problem to the backlog:

> P5. Representation of vectors: Vector should remember its own dimension; use of vector should follow the convention in mathematics.

Round 4. Problem P5 was tackled since it is the only one remaining. The program of round 3 was refactored to change the representation of vector from using C array to using C++ object. Here, the instructor carefully reviewed the structure of the program with the students and listed the following tasks (*Devising a Plan*):

T0. Learn class construct, constructors, destructors, and other member functions.
T1. Code up the vector class.
T2. Test vector.
T3. Change the function inner product.
T4. Change the unit test of inner product.
T5. Change extract vector from string.
T6. Change the unit test of extracting vector from string.
T7. Change get vector from user.
T8. Change main.
T9. Test main by running the program.
T10. Clean up in the reverse order of changes.

As can be seen, the change was pervasive. The tasks included learning object representation (T0), implementation, testing, and rework. The instructor also took this opportunity to demonstrate unit-test driven development in class, by doing T2 before T1, T4 before T3, and so on (*Carrying Out the Plan*; *Selecting and Sequencing*). In *Looking Back*, the instructor reviewed the program (shown below) with the students and compared with the previous code:

```
6   int main()
7   {
8       bool cont = true;
9       while (cont)
10      {
11          try {
12              Vector vec1 = getVectorFromUser();
13              Vector vec2 = getVectorFromUser();
14              double ip = computeInnerProduct(vec1,vec2);
15              std::cout << "Inner product is:" << ip << std::endl;
16          }
17          catch(char const * s) {
18              std::cout << s << std::endl;
19          }
20          cont = promptUserToContinue();
21      }
22      std::cout << "Terminating program..." << std::endl;
23      return 0;
24  }
```

With the object representation, vector now appeared in only three different locations:

creating vectors (lines 12 and 13)

passing vectors to computeInnerProduct (line 14)

Comparing with its predecessor shows that object representation was more concise and easier to understand than the array representation.

Subsequent examples. The second problem was to compute linear transformation (which is represented in a matrix) to vectors. The third problem was to compute the

area and perimeter of a convex polygon. Each of them was completed in three rounds applying *How To Solve It*. Both problems were subject to the constraint to reuse (and extend if necessary) the vectors of the first problem. In the case of polygon, the vertices of the polygon were represented in vectors. The computation of area and perimeter depended on sorting the vertices in the counterclockwise order with respect to a reference point inside the convex polygon. The sorting was done by making use of the STL function *sort*. The sizes of the programs of the three examples increased with rounds as shown in Table 1. Note that the last program released contains 1678 lines of code.

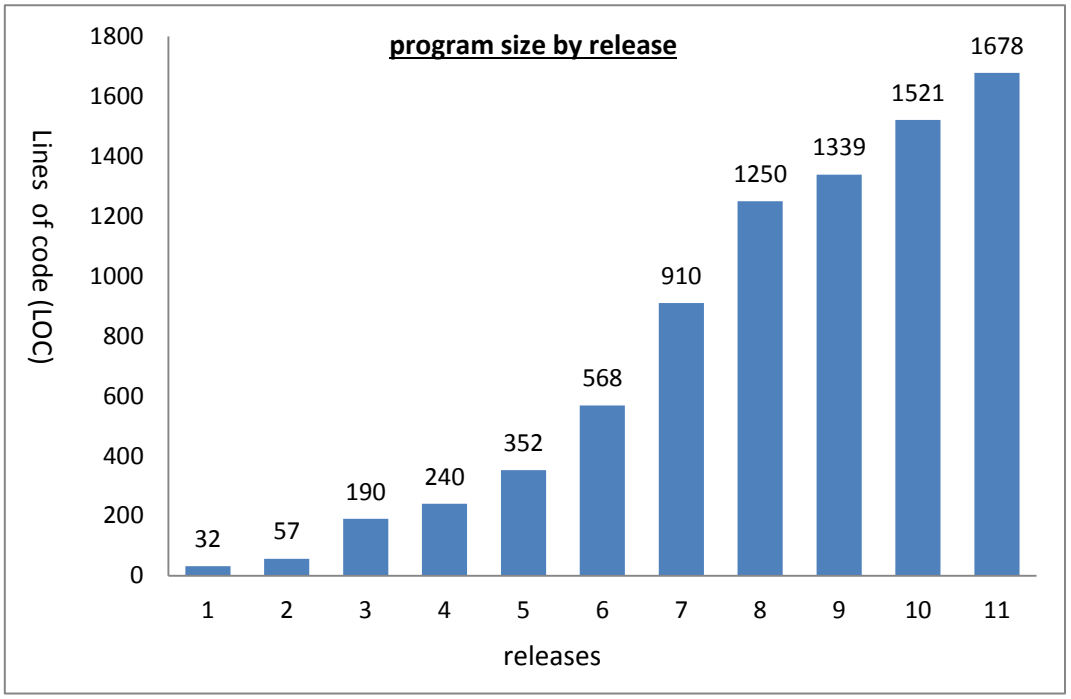


Figure 1. Sizes of programs (lines of code) in the eleven releases

Results. The students were polled for their responses to the questions regarding the specific way the course was conducted using *How To Solve It* at the completion of the course. The result can be found in Appendix B. Eight out of twenty-two students responded in the survey. Overall, the result seemed positive. It is interesting to note the respondents agreed or strongly agreed that they were comfortable in handling a program of 1000 lines or more after taking the course, which was in sharp contrast with their indication of being comfortable in handling a program of size 300 lines or less prior to taking the course.

The actual allocations of time to methodologies and engineering practices, language and libraries, and object orientation were 19.5%, 37.8% and 42.7%, respectively. The instructor felt the allocations were appropriate. In particular, the

experience shows that it is possible to cover *How To Solve It* and engineering practices within less than 20% of the lecture time. Thus, the instructor achieved what he had intended in covering the essential methodologies and engineering practices. Note that a significant number of respondents felt that more coverage of language and libraries was necessary. In contrast, they felt that OO coverage could be reduced. Given that most of the students had not programmed in the previous year before taking this OOP course, the response was not surprising. Learning language features remains a challenging task. Lastly, the students felt that they worked hard in this course.

4. Conclusion

This paper presents the four steps of *How To Solve It* in the form of patterns. They have been applied in teaching numerous offerings of OOP as a second programming course in the past few years. Our experience has shown that *How To Solve It* can be an effective way of teaching such a course. The exit survey conducted at the end of the most recent offering appears to reaffirm our claim.

In continuing to expand the use of *How To Solve It* to other programming courses, we plan to research in greater depth on Polya's classic treatise. To us, the Polya's *How To Solve It* is a pattern language that educators can find inspirations in. Also, it should be interesting to see how the *How To Solve It* patterns can be applied together with the existing pedagogical patterns [Ber2000] and problem-based learning [Kay2000].

Acknowledgements

We thank our shepherd Joe Yoder for his constructive comments which help improve this paper into its current form, in particular, the inclusion of the pattern *Selecting and Sequencing*. This work is supported in part by Auburn University.

Reference

[Ale1964] Alexander, Christopher. Notes on the Synthesis of Form. Harvard University Press, 1964.

[Bec2003] Beck, Kent. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[Bec2004] Kent Beck and Cynthia Andres. Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional. 2004.

[Ber2000] Bergin, Joseph. "Fourteen Pedagogical Patterns." *EuroPLoP* 2000.

[Che2009] Chen, Chien-Tsun, et al. "Delivering Specification-Based Learning Processes with Service-Oriented Architecture: A Process Translation Approach." J. Inf. Sci. Eng. 25.5 (2009):

1373-1389.

[Che2014] Cheng, Y C, "Applying How To Solve It in Teaching Object-Oriented Programming and Engineering Practices." AisanPLoP 2014.

[Che2014-2] Cheng, Y C, "How To Solve It: CPP." http://htsicpp.blogspot.com/. Accessed 2015/1/6.

[Cod2014] Code::Blocks, An open source IDE for C, C++ and FORTRAN. Available from http://www.codeblocks.org/.

[Coh2004] Cohn, Mike. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[Cor2001] Cormen, Thomas H., et al. Introduction to algorithms. Vol. 2. Cambridge: MIT press, 2001.

[Jac2001] Jackson, Michael. "Problem frames: analysing and structuring software development problems." (2001).

[Kay2000] Kay, Judy, et al. "Problem-based learning for foundation computer science courses." *Computer Science Education* 10.2 (2000): 109-128.

[Lar2012] Larman, Craig. "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3/e." (2012).

[Pat2008] Patton, Jeff. "The new user story backlog is a map." *http://www. agileproductdesign. com/blog/the_new_backlog. Html.* (2008). Accessed 2015/1/3.

[Pol1957] Polya, George. *How to solve it: A new aspect of mathematical method*. Princeton University Press, 1957.

Appendix A: The umbrella pattern *How to Solve It*

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. In so doing, side-by-side coverages of OOP and engineering practices are necessary. Since the time available for classroom teaching is limited, several conflicting forces are at play in such a context. Derives from George Polya's classic on mathematical problem solving heuristics [Pol1957], *How to Solve It* is a pattern that has been used to resolve the conflicting forces in developing and using complex and long-running programming examples for use in classroom teaching and learning of OOP and engineering practices [Che2014].

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 of writing programs with sizes up to a couple of hundred lines of code. They also have limited knowledge of engineering practices that are generally useful in 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 lectures in class and practice outside class.
- Detailed coverage of language features can be time-consuming.

Solution: Prepare long running examples for use in class and guide 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*.

Consequences:

1. Iterative and incremental development is taught and learned through long running examples and homework assignments that build on each other.
2. Learning of language features, object-orientation, and engineering practices takes place in the specific context of solving the problem on hand.
3. Classroom coding with student participation.
4. In-depth learning now becomes the student's responsibility.

Appendix B: Exit survey of a course at Auburn University taught with *How To Solve It*

The pattern *How To Solve It* and the supporting patterns described in this paper were used in the course "COMP3000 *OOP for Scientists and Engineers"* at Auburn University in fall, 2014. The students were polled *after* the announcement of their semester grades. Sixteen items targeting measurement of the consequences of the pattern *How To Solve It* are included. In the order that the items appear, the survey includes two items on program size (consequence 1), four items on allocation of lecture time to topics (consequence 2), three items on classroom coding during

lectures (consequence 3), three items on the effectiveness of applying *How To Solve It* in a long running example (consequence 1), two items on the engineering practices of unit testing and refactoring (consequence 2), and two items on student efforts (consequence 4).

Classroom coding 3

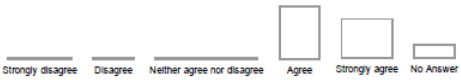
8 attempts

The amount of student participation in classroom coding is

How To Solve It

8 attempts

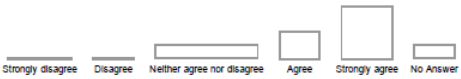
I find *How To Solve It* -- the iterative four-step method for solving programming problems -- helpful.



Blog

8 attempts

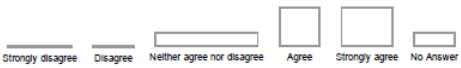
The articles on the blog *How To Solve It: CPP* are helpful.



Long running example

8 attempts

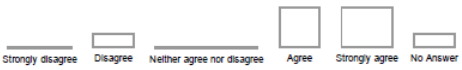
The long running programming examples that span multiple lectures are helpful.



Unit testing

8 attempts

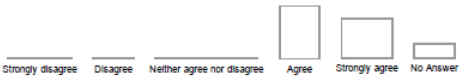
The early introduction to unit testing (in the second week) is helpful.



Exceptions

8 attempts

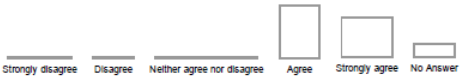
The early introduction to exceptions and exception handling (in the second week) is helpful.



COMP 3000 1

8 attempts

Overall, COMP 3000 is challenging.



COMP 3000 2

8 attempts

Relative to the other courses I have taken, I worked hard for COMP 3000.