Embedding Software Engineering Disciplines in Entry-Level Programming

Lung-Lung Liu llliu@mcu.edu.tw
Associate Professor, International College
Ming Chuan University
Gui-Shan, Taoyuan County, Taiwan, ROC 333

ABSTRACT

Software engineering disciplines can be embedded in entry-level programming assignments as a very basic requirement for teachers to the students in the classrooms and mentors to their apprentices in the offices. We are to use three examples to demonstrate how easily some of these software engineering disciplines can be embedded, and we will then prove that they are helpful for quality and productive software development from the point of being with "no source code modification" when some requirements are changed. In fact, convergence can be confirmed even there have been these changes. If the entry-level programming works are with software engineering disciplines, then the total software development effort should be decreased. With this concept in mind for project managers, actually, there are simple refactoring skills that can be further applied to those programs already coded.

Keywords: Software Engineering Practice, Preventive Maintenance, Requirement Change.

1. INTRODUCTION

We had the experience to give software engineering courses to computer science (junior/senior) students in the campus and software company (programmer) employees in the industry. They are with good programming language skills, such as the handling of syntax details by using Java or C#; but to most of them, software engineering is just like extra requirements to their previous works. They have to change even from their minds. The reasons are clear. The conventional education strategy in computer science is to divide the software courses into programming groups and software engineering groups, and the programming ones are the mandatory. Having studied the programming related courses, the students may want to select the software engineering related ones. That is, when people are to take a software engineering course, their personal programming styles have been set already. However, the styles may be improper for software development if necessary attentions were not paid.

Why don't we embed some of the software engineering disciplines in those programming related courses, especially the entry-level ones? In the following sections we are to demonstrate this by providing three examples: the popular "hello, world" program [1] introduced as the first trial in various programming languages, the state transition [2] handling program, and the sorting [3] program. We will show that some software engineering disciplines can be easily embedded in these entry-level programming exercises. These disciplines are for "no source code modification" to possible requirements change, which are usually to happen. In other words, a considerable programming style (with software engineering disciplines) can help reduce the chance to modify

the source code. It is true that if code need not be modified when there is a requirement change, then there is the higher possibility for quality and productive software.

The theoretical background of the disciplines is preventive maintenance [4], or a convergence software process, which makes sure that consecutive processes can really approach to the target. We will discuss this after the demonstration of the examples. The other software engineering technology can be directly applied here is refactoring [5]. It is for programs already coded by students or junior employees. To the teachers and managers, asking them to do the refactoring works is a challenge. Nevertheless, automated skills but not labor-intensive routines should be considered. When the students and programmers are used to embedding software engineering disciplines in their daily coding works, Personal Software Process [6] is then significant.

2. THE "HELLO, WORLD" EXAMPLE

The program titled "hello, world" has been the first trial in learning different kinds of programming languages for years. Although there are versions of the program, the major function is to output the words: hello, world. In the following, we use a common pseudo code to specify the program. A typical and simple version may look like this:

```
function hello_world()
begin
        print("hello, world')
end
```

From a programmer's point of view, it is well done. However, even the program is titled as hello_world, a customer or stakeholder may request a change for alternative output of the words: hi, globe. They are the same, technically in programming skills, but they are actually different programs since the code should be modified. There are risks to introduce human errors in the modification processes.

Literals should be avoided in programs, and variables and parameters are suggested in general. We skip the use of variables and go straight for the use of parameters. The new version of the program may now look like this:

```
function hello_world()
begin
        print(get_whatever_requested())
end
```

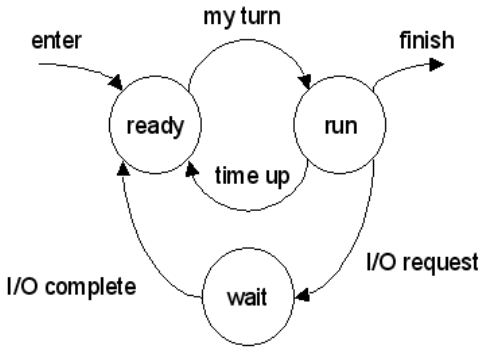
The nested function get_whatever_requested() is flexible and powerful. It can be another system supported function just like the print(something) function. The programmer is then waiting there, for all possible words requested as the desired output, since there is no need to modify any of the code. (Actually, no re-compilation is necessary.)

There are ways to let the system know where to get those whatever requested by providing a customizable profile, and the contents may indicate the input source such as the keyboard, a file, a database entry, or those through a network. The format of the input data can be further specified, such as whether the requested words are in English or Chinese.

In the past, we seldom saw students (or junior programmers) go this way at their earlier programming learning stages. Their styles were pure "hello, word" by referencing the books of introduction to programming languages. Although the programming (development) environments have been greatly improved since Java and C# were introduced years ago, the design of them together with software engineering basics is still blocked.

3. THE STATE TRANSITION HANDLING EXAMPLE

The handling of state transition is another typical exercise for computer science students and business application oriented junior programmers. The processes in a multitasking operating system is basically synchronized by a scheduler, and the processes are well controlled to be with ready, running, or waiting states. The current status of a running business can also be managed by a control system, and each of the possible sales cases (can be specified by using sales forms) can be associated with a current state, such as initial, in review, waiting for approval, or final. The following is a state transition diagram of processes in an operating system, and it is actually the requirement for the implementation of the handling program:



According to the diagram, a direct program with a lot of conditional statements is obtainable. However, we know that conditional branches and expressions in conditional statements are with high risk to introduce errors in a program. Furthermore, when the customer or the stakeholder once raised a change, the effort of modifying the code will be with even higher risks of new errors.

An advanced approach is to use the state transition table, which is equivalent to the state transition diagram but is much more precise for programming:

event / state	my turn	time up	I/O request	I/O complete
ready	run			
run		run	wait	
wait				ready

- A newly entered process is in the ready state
- A finished process will leave

The controlling of state change is now with no conditional statement coded, since the change can be determined by checking with row index (the current state) and column index (the input). The cell with proper indices in the table tells the next state. In addition, the size of the table doesn't need to be fixed. Or, the number of rows and that of columns can be assigned outside of the code, hence again, like we have mentioned in the previous example, they and the whole contents of the table can be specified in a customizable profile.

We experienced so many cases that students' (and employees') programs are designed with long and deep conditional statements, but it seemed that some of them did enjoy this style. Actually, a

professional state transition handling mechanism (software) is with only limited size (lines of code), and it is almost always reusable. The programming books should guide the readers how to design the contents in a state transition table but not how to code according to a diagram.

4. THE SORTING EXAMPLE

To solve the sorting problem is a typical exercise to beginning programmers, especially to computer science students whose teacher wanted them to get into algorithms as early as possible. Usually, after students have tried the bubble sort and quick sort skills, the job is done. However, in practical programming concerns, there are much more. The following is the list of weekdays in their conventional sequence:

Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

What is the result if we let a common sorting program to run with this list as input? If no further information is provided, the output is this:

Friday
Monday
Saturday
Sunday
Thursday
Tuesday
Wednesday

It is somewhat funny since common users will get confused with the result, but the computer science students will support the result with explanations in technical terms. A tentative conclusion may be that the computer is not as clever as human beings, because it does not understand what weekdays really are. However, the actual problem is the sorting program but not the computer. The truth is that the program is lack of the thoughtful considerations of user friendly.

A proper sorting program should be able to handle different data types, formats, languages, and even semantics. Data definitions, data dictionaries, and even guidelines can be associated with the program as changeable supports, but the code can be kept unchanged at all. Although we have not seen a universal sorting program developed by the students and employees until now, it is encouraged for them to think this way. It is the embedding of software engineering disciplines in entry-level programming.

5. CONVERGENCE

The three demonstrated examples should have indicated some of the software engineering disciplines such as: no literal in the code, simplifying the code whenever it is possible, and real information hiding. There may be debates, although they are just examples for better understanding. In the following, we are to prove that they are helpful for quality and productive software development from the point of being with “no source code modification” when some requirements are changed, and it is easy to do so.

Fact 1. Some requirement change can be handled by changing the contents in a provided user-customizable profile.

A typical example of this is the use of the win.ini file in the popular MS Windows systems. When a user is to change his/her desktop's background color, a setting in the profile satisfied the request. Either a direct text editing or a mouse clicking through graphical user interface completes the setting.

Fact 2. A program is composed of data sections and instruction sections. No source code modification of a program means that there is no change, neither in the data sections, nor in the instruction sections.

Data sections are declared as variables and literals (such as specific numerals and strings), and instruction sections are specified as statements. The instructions refer to literals and current values of variables, and then new data values of variables as computation results are generated.

Fact 3. The total number of errors in a program will not increase unless the program (source code) has been modified.

There is the number as the total number of errors in a program, although it is very hard to find the number. (Usually we use testing record to estimate the number.) However, if the program has been modified, then the relationship between the new number of errors and the original one is unclear (although both of the two numbers are unknown). What we are sure is this: If there is no change in the code, then everything is still fixed as it was before.

Theorem 1. For some of the requirement change, no source code modification can be achieved by replacing literals with contents in a user profile.
[Proof] Let R be the requirement. A source code C is programmed to first read from the user profile F for a value, and then assign the value to variable V in C as the requirement. R is originally in F, and C can get R for V. According to Fact 1, when there is a requirement change from R to R', we first replace R by R' in F, and we then have C perform the same process for V. R' is now the value assigned to V. However, according to Fact 2, there is no source code modification in C.

Theorem 2. No source code modification is helpful for quality and productive software development.
[Proof] According to Fact 3, the total number of errors will not increase if there is no source code modification in a program. For quality software development, this shows a convergence (or at least, non-divergence) in the programming versus requirement change processes, since the trend of introducing new errors does not exist. For productive software development, it is clear that the language translation procedures such as the recompilation of the modified program, the make of a new load module, and the adjusting for a new run environment, can be avoided. Hence, it is with higher productivity.

Theorem 3. It is doable to embed software engineering disciplines in entry-level programming.
[Proof] There is no special programming skill for students or junior employees to embed software engineering disciplines in their works. According to the three examples demonstrated, it is easily doable.

6. REFACTORING

Refactoring is the process to modify a program but with no functional change. It was introduced recently for advanced software development approaches such as extreme programming techniques. However, for entry-level programmers, the concept is applicable. In short, it is not the work for requirement change from the customers and the stakeholders, but it is the work for a

student or junior programmer to enhance his/her current code. The purpose, from a software engineering point of view, is also for quality and productive software development.

Most of the cases for students and junior employees in their programming experience indicated that, after a supposed to be correct output has been obtained, the programs will not be modified. Actually, the teachers or managers should ask or guide the students or junior employees to continue their works for no source code modification to some of the requirement change, as was discussed before. The theorems say that the literals in their programs can be easily replaced by contents in a user customizable profile. If this has been the discipline naturally embedded in the programming works, then they are on the right software engineering way.

Automated skills but not labor-intensive works are suggested for refactoring. For example, a first exercise to entry-level programmers may be the scanning of literals in a program, and then the second step is to properly replace the code. The refactoring process is to modify the code where a literal is used by function calls (reading an external user profile) that can return a value equivalent to the original literal. Actually, the use of literals in a specific programming language can be handled in compilers as an optional feature. Hence, the automated skills for refactoring can be applied.

7. CONCLUSIONS

Software engineering disciplines can be easily embedded in entry-level programming exercises as requirements. We have tried to use three examples to demonstrate how software engineering disciplines can be embedded, and we proved that they are helpful for quality and productive software development. No source code modification when some requirements were changed is the theme. The theoretical background of the disciplines is preventive maintenance, or a convergence of a software process, which makes sure that consecutive process steps can really get approach to the target. In fact, convergence can be confirmed even there have been requirement changes, since the code is not changed. If the entry-level programming works are with software engineering disciplines, then the quality of software development should be with well control, and the total software development effort should be decreased. To be more aggressive, there are simple refactoring skills that can be further applied to those programs already coded.

Our results are logically significant, and they are also practical. For example, there have been the studies on software maintenance and software reusability, such as (1) the handling of maintainability, especially the changeability [7], and (2) the metrics of reusable code [8]. Our results may indicate that (1) the handling effort of changeability can be easier, no matter the design is aspect oriented or not, and (2) the CRL LOC (i.e., the Component Reusability Level based on Lines of Code) is actually 100%. The reason is obvious since there is no code modification. (However, the proof is beyond the scope here.) The other example is with the discussion of emphasizing on preventive maintenance in the introducing of a software process to an organization [9]. The experience also indicated that the case of individual programmers with good software engineering disciplines is a key successful factor .

The future work of this study is on the requirement analysis of a practical universal sorting program. No matter what the requirement change is, the code of the sorting program is not to be changed. Although we have collected many requirements from groups of end users, it is still on going. One typical requirement is to provide a sorting program for a list of postal addresses in Chinese.

8. REFERENCES

1. Brian W. Kernighan, *"Programming in C: A Tutorial,"* Bell Laboratories, Murray Hills, N.J., USA, 1974

2. Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne, *"Operating Systems Concepts,"* Seventh Edition, John Wiley and Sons, Inc., Ch. 5, 2005

3. Donald. E. Knuth, *"The Art of Computer Programming, Volume 3: Sorting and Searching,"* Second Edition, Addison Wesley, Ch. 5, 1998

4. Roger S. Pressman, *"Software Engineering: A Practitioner's Approach,"* Sixth Edition, McGraw-Hill, Ch. 31, International Edition, 2005

5. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts, *"Refactoring: Improving the Design of Existing Code,"* Addison Wesley, Ch. 8, 2000

6. Watts Humphrey, *"PSP, A Self-Improvement for Software Engineers,"* Addison Wesley, Ch. 1, 2005

7. Avadhesh Kumar, Rajesh Kumar, and P S Grover, *"An Evaluation of Maintainability of Aspect-Oriented Systems: a Practical Approach,"* in International Journal of Computer Science and Security, Volume 1: Issue (2), pp. 1~9, July/August 2007

8. Arun Shamar, Rajesh Kumar, and P S Grover, *"Managing Component-Based Systems with Reusable Components,"* in International Journal of Computer Science and Security, Volume 1: Issue (2), pp, 60~65, July/August 2007

9. Lung-Lung Liu, *"Software Maintenance and CMMI for Development: A Practitioner's Point of View,"* in Journal of Software Engineering Studies, Volume 1, No. 2, pp. 68~77, December 2006