

Teaching Tip

CFC (Comment-First-Coding) – A Simple yet Effective Method for Teaching Programming to Information Systems Students

Arijit Sengupta

Information Systems and Operations Management
Wright State University
Dayton, OH 45435, USA
arijit.sengupta@wright.edu

ABSTRACT

Programming courses have always been a difficult part of an Information Systems curriculum. While we do not train Information Systems students to be developers, understanding how to build a system always gives students an added perspective to improve their system design and analysis skills. This teaching tip presents CFC (Comment-First-Coding) – a method for assisting students with information systems design and development tasks where a significant portion of the goal is to actually build the system using a programming language and development environment. CFC uses a scaffolding strategy for building programs where the using the comment construct of the programming language. In CFC, the first step students perform is to describe the programming task via plain English (or any other natural language) inside comments. The CFC process strategically and incrementally builds on this method to gradually add functionality and complexity to the program, while allowing the student to compile and test every individual step. In multiple offerings of a sophomore level data structures course, this method has provided evidence of improved student performance.

Keywords: Programming, Application Development, Scaffolding, Coding, Assisted Program Design.

1. INTRODUCTION

Building information systems is a significant component of any Information Systems curriculum. There are two aspects of building information systems – (a) design and (b) development. Most Information Systems curricula include a Systems Analysis and Design course, with potential follow up courses such as Object-Oriented Design, Database Design, etc. While colleges of business do not aim to produce programmers, the knowledge of programming provides business analysts with a better insight towards the actual efforts needed for developing an information system solution. Besides, the accrediting bodies for the colleges of business realize the necessity for application development knowledge in an Information Systems curriculum. Specifically, the ABET Information Systems accreditation requirements include the knowledge of at least one programming language in the Information Systems programs (ABET, 2008). The IS2009 curriculum, a joint venture between the Association of Computing Machinery and Association of Information Systems also recommend one or more application development courses, and even an application development track in the IS curriculum (Topi, et

al., 2009), demonstrating clearly the importance for IS faculty to teach application development topics in courses.

One of the most difficult aspects in teaching application development strategies to students is to help them move from a problem description to logic, and finally from logic to programming code. To reduce the complexity of the programming structure, different types of design methodologies have been introduced, such as the Object-Oriented Design (OOD) and Aspect-Oriented Design (AOD). A number of visual tools exist that help application developers build the skeletal structures of their applications, (e.g., Rational Rose Data Modeler (<http://www-01.ibm.com/software/awdtools/developer/datamodeler/>)). Many diagramming tools such as DIA (<http://live.gnome.org/Dia>) and Microsoft Visio (<http://office.microsoft.com/en-us/visio/>) also include similar capability. While these tools can be used to create the object structure, they do not help with the final phase of translating specification into logic and then into code.

Application designers use logic design methods such as Flow charts (IBM, 1969) that to capture program structure, including loops, decisions, procedures and branches. However, a flow chart for even a moderately complex program can become exceedingly complex. Tools such as

FlowC (Gill, 2004) help reduce some complexity by reducing the drawing overhead of flow charts. Nassi and Shneiderman (1973) developed the Nassi-Shneiderman diagram (NSD) that allowed program structures to be represented in a more compact form. Scandura (1990) generalized and extended NSD using Flowforms, and enabled visual programming by allowing the development of programs via a semi-textual interface. Stone (1987) proposed various instructional techniques for using Flowforms in introductory Computer Science instruction.

Visual and guided programming has shown to improve programming efficiency and code performance, while preserving program comprehension (Naharro-Berrocal, Pareja-Flores, Urquiza-Fuentes, & Velazquez-Iturbide, 2002). Visualizations are especially useful for the purposes of debugging (Baecker, DiGiano, & Marcus, 1997) and other post-development tasks. While several design tools that provide visual methods to complex development tasks are in existence, most of these tools require additional costs or learning time that adds to the difficulty of designing courses with a significant development component. While there are many tools that help programmers design and create applications, students learning the concept of programming need a technique that help in the process of gradually move from specification into logic, and finally from logic into actual code. This is the motivation for the teaching tip presented in this paper, which we introduce next.

2. CFC (COMMENT-FIRST-CODING)

In order to gradually build a structure, construction workers need a form of support to develop the structure from the specification and raw materials. In construction, such structures are called “scaffolding”. In instructional design, the concept of scaffolding has been very successfully used in helping students develop language skills such as reading and writing (Applebee & Langer, 1983). Scaffolding in the form of screencasts (video screen captures of visual development tools) have been shown to be effective means for teaching object-oriented design concepts (Lee, Pradhan, & Dalgarno, 2008). The issue with scaffolding is that the assistive methods are temporary and are discarded during the final development stage. We expand on the success of the scaffolding theory in a method we call Comment-First-Coding, where comments or code documentation fragments are used as the assistive tool, and developed systematically over levels, but are actually incorporated into the final result as well.

In the CFC method, the students are taught to write code by starting with only comments, and incrementally adding functionality to it. Instructors should prohibit students from writing any code until they have thought the whole design task through, and documented their understanding of the task by writing down the different steps of the task in English (or their native language, if the development environment allows it). Once the students have described the whole system (and most likely, have revisited previous steps as they think through subsequent steps), they can then start incrementally adding further semantics to their program by including more code, more comments, or additional structures. To keep things consistent, I will assume that the

students are using an object-oriented language like Java, although the method can be used for any programming language in which application logic needs to be implemented using program code.

We now present the details of the CFC technique. As in scaffolding, the methodology is developed in stages, starting with an object-oriented (or module-based) design, followed by logic development, structure development, and finally code development.

STAGE A. Pre-CFC

1. Design the application using an object-oriented design method or tool. The class structure should already be designed before CFC. This can be done by an object-oriented design tool like Rational, or just by hand. CFC is not a method for designing application architecture, but for designing logic for individual programming tasks.
2. Identify one of the tasks that need to be implemented. This can be a method in a class with a clearly defined and understood semantics. The task should have clear specifications as to what will be available as input, and what should be generated as output.

STAGE B. LOGIC DEVELOPMENT

3. This is the first actual step in the CFC process, where the entire logic for the current method being developed is written in English. However, instead of using a free flowing paragraph, the students describe the program in logical steps. No programming is needed in this step, except for the class and method declarations. The CFC-3 comments go inside the body of the methods that are to be implemented.

STAGE C. STRUCTURE DEVELOPMENT

4. For each step created in Step 3, add basic code blocks only to build the overall program structure. Such code blocks should only include the following:
 - a. Variable declaration and initialization (with descriptive comments and which steps they will likely be used)
 - b. Basic program structure (if-else, for/while loops, switch-case). The internals of the structures should be empty, only the required parts of the structures should be completed in this step.
 - c. Ensure that the comments are properly enclosed in the structures in which they belong. Expand the comments to provide additional details if necessary.
5. For any step that requires sub-steps, repeat steps 3 and 4, until all steps are fully expanded.

STAGE D. CODE DEVELOPMENT

6. Finally, fill in the empty structures, starting from the most obvious (and easiest) steps and pre-requisite steps if any. Compile and test after each section.

The above four stages and six steps provide a rigorous method for developing programs using any development environment or source code editor. I typically refer to a program implementing the above steps with the step number, such as CFC-3, CFC-4, etc. A program is CFC-3 if no part of the program implements CFC step 4. Instructors looking for

a more compact option may also use the stages as CFC-A, CFC-B, CFC-C and CFC-D.

3. ADVANTAGES OF CFC

There are several advantages for using CFC:

1. First of all, CFC is independent of programming languages. Every programming language has a comment syntax, and CFC takes advantage of this aspect of the programming language. The comments use only natural language text and no proprietary graphical constructs, and hence do not require any specialized tool or software. Any IDE or source code editor can be used for CFC.
2. Comments are guaranteed to compile – students only need to worry about comment markers. While this may sound trivial, in my experience many novice programmers use a linear approach to programming, and do not attempt to compile the code in steps. As is common with compilation errors, one error may lead the compiler to generate several error messages, so often students are distracted by multiple error messages. Since comments never generate error messages, the step-by-step approach in CFC ensures that the students can compile their work at each step and ensure no new compilation errors are found, and if the code does not compile, they know immediately which step caused this error.
3. Once finished, the source code is well-documented, ensuring readability and reusability of the code. The comments can also be used to generate code documentation using a tool like Javadoc™ (SUN, 2002).
4. CFC allows different possibilities for assigning programming tasks to students. Students may be asked to start from Step 1 or possibly an intermediate step (such as Step 3 or 4). For a complex programming project, the instructor may provide the students with CFC-3 (and possibly CFC-4 code) that the students have to complete.
5. Results from an initial use of CFC suggest that more students submit code that actually compile, so they can be tested easily by the grader.
6. Since students only need to write in English in CFC-3 (and much of CFC-4), often this infuses more creativity in the students, resulting in potentially interesting solution strategies.

4. USING CFC IN A COURSE PROGRAMMING PROJECT

In order to use CFC effectively in a course, the instructors will need to add a little more rigor to programming assignments. Simply stating the task will likely not force the students to use CFC, and they may be tempted to start programming the way they are used to, and add comments later, in order to fulfill the requirements of the task and giving a false illusion to the grader that they did follow the CFC guideline. To paint a complete picture, I will describe an example of an assignment that I have used in a 200 level data structures course, where students need to use stacks.

The following assignment asks the students to build a non-recursive program that uses a stack to find goal

conditions in a virtual game tree that can solve many different types of problems. The task is shown in Figure 1.

“You are going to design a simple maze game. A maze can be thought of as an $m \times n$ grid of cells. Each grid cell is like a room with 4 sides: north, east, south, and west. A side of the room is either a complete wall or a wall with an open door (entry way). You can assume that the four walls can be represented as North, South, East and West. One or more of these walls must have a door (none of the doors will lead outside the maze, so every door will lead to one other room in the maze. Assume that there is light in each room so you can see. A mouse is initially placed in some room and a slice of cheese is placed in another (far from the mouse). Can you help the mouse get to the cheese?”

Figure 1. The MazeGame Task

To develop the solution for this problem, the students first need to complete Stage A (CFC steps CFC-1 and CFC-2) where they develop the object-oriented design for the solution, with a Maze data structure, a MazeReader class that reads the maze from a text file representation, and the MazeGame class that plays the game. In the MazeGame class, they build a method (say, called, *run*) that actually runs the solver. This is where the actual CFC process CFC Stage 2 starts with the CFC-3 comments. As mentioned above, the comments should be written with explicit steps describing the logic entirely in English with no programming needed at this stage except for the method declaration. Details of these stages for this example are described below.

4.1. Developing CFC Exercises

CFC exercises need more up-front work from the instructors. Instead of assigning a single programming assignment, the assignment needs to be developed in stages. The stage deliverables can follow the CFC stage levels as described earlier, with the first deliverable being CFC-3 (Stage B), followed by CFC-4 and CFC-5 (Stage C) and finally the completed program (Stage D).

In the first step, the instructor provides the students with the problem statement, and asks the students to build the CFC-3 solution for the problem (See Figure 2 for the CFC-3

```
01 public void run(String filename) {
02 // Declare variables - what do we need to remember?
03 // Strategy
04 // Step 1. Read in the maze and populate a 2D array of Rooms
05
06 // Step 2. Create an instance of a stack.
07
08 // Step 3. Place a Location or a Room in the stack - it doesn't really matter
09 // which one you place, but you should create the stack accordingly.
10 // Place the Room/Location corresponding to the mouse position in the stack.
11
12 // Step 4. Now use the following strategy.
13 // While there is something in the stack
14 // Peek at the top room of the stack.
15 // If this room is not visited, do the following:
16 // Mark it to be visited.
17 // If the room location is the same as the cheese location, then you are done.
18 // display the route and exit.
19
20 // If not, find all the rooms that are accessible from this top room
21 // that are not visited
22 // (you can use a specific order, or random order - it does not matter).
23 // Push all the neighboring un-visited rooms in the stack.
24 // If the top room is already visited, pop it off the stack. Continue with Step 4.
25
26 // Step 5. If the stack becomes empty, there is no solution to the problem.
27
28 // Step 6. Find a way to show the solution - Hint - the solution is in the stack
29 }
```

Figure 2. A CFC-3 representation of the traversal strategy for the maze problem

solution for the Maze Game task, showing only the run method). In this stage, the instructor should ask the students not to be concerned about file structures, I/O and other auxiliary tasks. No part of the problem should be implemented at this point. The students should only identify the constituent steps in the code (CFC-3). The instructor should provide feedback to the students regarding their logic depicted in CFC-3 before they begin working on the next stage.

4.2. Phase 2 – Completing the Exercise

In the next phase of the assignment, the instructor asks the students to build on the CFC-3 they created in the first stage, to include variable declarations and basic program structures and sub-structures (CFC-4 and CFC-5). If necessary, the instructor may also provide his or her own CFC-3 solution for students to start with, ensuring all students start in the same place for this part of the exercise, regardless of their performance in the first step. Figure 3 shows the CFC-3 code created for part of the Maze game strategy. The changes are highlighted. Notice how the required variables are declared, and the structures are created. Also note that one or two lines of comments are translated into a few lines of code, but each code segment only implements the comment immediately preceding it. The students are encouraged to only write code that is described in the comment before it.

Figure 4 shows the CFC-5 expansion (substeps) for the maze game strategy. CFC-5 is essentially repetition of CFC-3 and CFC-4 steps for any sub-steps.

In the final step, the students complete the code and build the fully completed CFC-6 solution, which is a completed solution of the original problem with fully

expanded documentation describing the logic and the implementation of the logic for each step. Figure 5 shows the CFC-6 expansion for a part of the Maze game problem. This is the final code, although the amount of change from the previous step is quite manageable. The changes from the previous step are highlighted in Figure 5.

5. RESULTS OF USING CFC IN A COURSE

I have used CFC successfully several times in my offering of a 200 level undergraduate data structures course. Although the course is numbered as a 200 level course, many students often delay taking this course until their junior or senior years because they know of the programming aspects of the course. This results in a fairly diverse group of students in the class, making it harder for an instructor to teach at an appropriate level. Using CFC, I have found that students in all different levels are able to stay with the pace of the course and not fall behind. Since the initial CFC-3 or CFC-4 task is an actual graded assignment, students get feedback from their first attempt, and can correct it, or actually look at or use the CFC-3 or CFC-4 code provided for the follow-up task. Table 1 shows data from three offerings of this course. In the first offering, CFC was not used. I introduced the concept of CFC out of frustration of the students giving up and submitting code that did not even compile. The first offering shown in Table 1 shows that as many as 40% of the students submitted code that did not compile correctly in Assignment 1, so an automatic testing could not even be applied.

In the second offering (term 2), I introduced students to the CFC concept midway in the class, and noticed a substantial

```
12 // Step 4. Now use the following strategy.
13 // While there is something in the stack
13a while(!mystack.empty()) {
14 // Peek at the top room of the stack.
14a Room top = null; // The top room
15 // If this room is not visited, do the following:
15a if (!top.isVisited()) {
16 // Mark it to be visited.
17 // If the room location is the same as the cheese location, then you are
17a if (top.getLocation().equals(cheeseloc)) {
18 // display the route and exit.
19
19a }
19b else {
20 // If not, find all the rooms that are accessible from this top room
21 // that are not visited
22 // (you can use a specific order, or random order - it does not matter)
23 // Push all the neighboring un-visited rooms in the stack.
24 // If the top room is already visited, pop it. Continue with Step 4.
24a }
24b }
```

Figure 3. CFC-4 expansion for lines 12-24 (Step 4) in Figure 2.

```
12 // Step 4. Now use the following strategy.
13 // While there is something in the stack
13a while(!mystack.empty()) {
14 // Peek at the top room of the stack.
14a Room top = null; // The top room
15 // If this room is not visited, do the following:
15a if (!top.isVisited()) {
16 // Mark it to be visited.
17 // If the room location is the same as the cheese location, then you are done.
17a if (top.getLocation().equals(cheeseloc)) {
18 // display the route and exit.
19
19a }
19b else {
20 // If not, find all the rooms that are accessible from this top room
21 // that are not visited
22 // (you can use a specific order, or random order - it does not matter).
23 // Push all the neighboring un-visited rooms in the stack.
24 // If the top room is already visited, pop it. Continue with Step 4.
24a }
24b }
```

Figure 4. CFC-5 expansion for lines 19b-24a in Figure 3.

change in the number of students submitting syntactically correct code – the percent of students submitting correctly compiled code increased from 57% in assignment 1 to 86% in the first cfc step of assignment 2, and to 100% in the final step. I in the third offering, i introduced cfc right from the beginning, and although the correctly compiling code were 85% and 90% in assignment 1, the percentage went up to 95% and 100% in assignment 2.

Readers, please note that this was not a rigorous user-study but just based on the outcome of student performance

with the use of a different teaching strategy. I intend to perform an actual study for effectiveness of this teaching method as a potential future work. The data represent just a compilation of scores after the courses were taught, and were not collected with the intention of eventually comparing the results, so I am not making a claim that other factors (such as experience and improved teaching skills) did not cause the slight improvement of the scores. However, the percentage of students with no compilation errors did significantly improve, which was one of the goals of this teaching strategy.

```

12 // Step 4. Now use the following strategy.
13 // While there is something in the stack
13a while(!mystack.empty()) {
14 // Peek at the top room of the stack.
14a Room top = mystack.peek(); // The top room
15 // If this room is not visited, do the following:
15a if (!top.isVisited()) {
16 // Mark it to be visited.
16a top.setVisited(true);
17 // If the room location is the same as the cheese location, then you are done.
17a if (top.getLocation().equals(cheeseloc)) {
18 // display the route and exit.
18b this.displaySolution(mystack) // Display the solution route
18c break; // We can get out of the loop
19
19a }

```

Figure 5. Final CFC-6 expansion for lines 12-19a in Figure 3.

Term	N	A1CFC		A1		A2CFC		A2		A3CFC		A3	
		%comp	Avg										
1	10	-	-	60	39	-	-	70	44	-	-	80	42
2	7	-	-	57	42	86	45	100	43	100	45	100	42
3	20	85	43	90	44	95	45	100	45	100	46	100	44

Table 1. Results from using CFC in a 200 level data structure course. (%comp: percent of students submitting code that compile without errors, Avg: average score out of 50)

Interestingly, students find CFC useful not only for the assignments in the course, but also for development work for their subsequent course projects, or even at work. The following excerpt from an email message from an ex-student suggests some evidence of effectiveness of this method:

“I remembered a while back you suggested doing comments first then coding (CFC). It's actually worked really well for this game tree problem. I implemented a hash table, using a hash of each sparse matrix as a key, but performance is still absolutely horrible. The raw complexity of this problem is n! - so for 17 values (a very small map) there are 355687428096000 iterations to test!”

6. CONCLUSIONS AND FUTURE WORK

CFC is a simple addition to any existing course that involves one or more development tasks. CFC is based on the well-accepted learning strategy of scaffolding. However, with CFC, the comments that are used for scaffolding purposes do not have to be discarded when the task is complete, but serves as a documentation strategy for the application. Students benefit by learning in stages, and accomplishing the task first by understanding and elaborating the logic, followed by the program structure, and finally the actual application code. The instructor also benefits from the fact that more students submit code that compile correctly, and hence can be tested via an automatic testing method. As in any learning-oriented strategy, this method requires more effort from the instructors' behalf, and instructors need to use a strategy similar to the FIDeLity strategy (Frequent, Immediate, Discriminating and Loving) for providing feedback after each stage of the tasks (Fink, 2003).

With CFC, the instructor adds one or more intermediate deliverables to every programming assignment, in which the students submit a completed program structure with no real code, but only filled with comments describing their implementation strategy. I believe that if this method is implemented with rigor, involving at least one CFC-4 assignment prior to all programming assignments, it will improve students' ability of writing code that is well-documented, free of compilation errors, and actually work. While I have not performed a rigorous user study to analyze the effectiveness of this strategy, results from initial use of this technique in a course suggests that this technique helps students think through their implementation and write code that can be tested properly. The rigor in the programming process enables students to identify areas of complexity and incremental development methods, thereby reducing common programming errors. A well-planned user study involving a control group of students should allow better evaluation of this technique.

7. REFERENCES

ABET (2008). Criteria for accrediting computing programs - Effective for evaluations during the 2009-2010 accreditation cycle. Retrieved from <http://abet.org/Linked%20Documents-UPDATE/Criteria%20and%20P/C001%2009-10%20CAC%20Criteria%2012-01-08.pdf>

Applebee, A. N., & Langer, J. A. (1983). Instructional Scaffolding: Reading and Writing as Natural Language Activities. *Language Arts*, 60(2), 168-175.

Baecker, R., DiGiano, C., & Marcus, A. (1997). Software visualization for debugging. *Communications of the ACM*, 40(4), 44-54.

- Fink, L. D. (2003). *Creating Significant Learning Experiences: An Integrated Approach to Designing College Courses*: Jossey Bass Higher and Adult Education Series.
- Gill, T. G. (2004). Teaching Flowcharting with FlowC. *Journal of Information Systems Education (JISE)*, 15(1), 65-78.
- IBM (1969). *Flowcharting Technique*: IBM Data Processing Techniques C20-8152-1.
- Lee, M. J. W., Pradhan, S., & Dalgarno, B. (2008). The Effectiveness of Screencasts and Cognitive Tools as Scaffolding for Novice Object-Oriented Programmers. *Journal of Information Technology Education*, 7, 61-80.
- Naharro-Berrocal, F., Pareja-Flores, C., Urquiza-Fuentes, J., & Velazquez-Iturbide, J. A. (2002). *Approaches to comprehension-preserving graphical reduction of program visualizations*. Paper presented at the Proceedings of the 2002 ACM symposium on Applied computing.
- Nassi, I., & Shneiderman, B. (1973). Flowchart Techniques for Structured Programming. *ACM SIGPLAN Notices*, 8(8), 12-26.
- Scandura, J. M. (1990). Cognitive approach to systems engineering and re-engineering: Integrating new designs with old systems. *Journal of Software Maintenance: Research and Practice*, 2(3), 145-156.
- Stone, D. C. (1987). A modular approach to program visualization in computer science instruction. *ACM SIGCSE Bulletin*, 19(1), 516-522.
- SUN (2002). Javadoc - The Java API Documentation Generator., from <http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javadoc.html>
- Topi, H., Valacich, J. S., Kaiser, K., Nunamaker Jr., J. F., Sipior, J. C., de Vreede, G. J., et al. (2009). *IS 2009 - Curriculum Guidelines for Undergraduate Degree Programs in Information Systems*: Bentley College.

AUTHOR BIOGRAPHY

Arijit Sengupta is an Associate Professor of Information



Systems and Operations Management in the Raj Soin College of Business at Wright State University He received his Ph.D. in Computer Science from Indiana University. Prior to joining Wright State, Dr. Sengupta served as faculty at Kelley School of Business at Indiana University and the Robinson College of Business at Georgia State University. Dr. Sengupta's current primary research interest is in the efficient use and deployment of RFID (Radio Frequency Identification) for business applications. His other research interests are in databases and XML, specifically in modeling, query languages, data mining, and human-computer interaction. He has published over 30 scholarly articles in leading journals and conferences, as well as authored several books and book chapters.



STATEMENT OF PEER REVIEW INTEGRITY

All papers published in the Journal of Information Systems Education have undergone rigorous peer review. This includes an initial editor screening and double-blind refereeing by three or more expert referees.

Copyright ©2009 by the Information Systems & Computing Academic Professionals, Inc. (ISCAP). Permission to make digital or hard copies of all or part of this journal for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial use. All copies must bear this notice and full citation. Permission from the Editor is required to post to servers, redistribute to lists, or utilize in a for-profit or commercial use. Permission requests should be sent to the Editor-in-Chief, Journal of Information Systems Education, editor@jise.org.

ISSN 1055-3096