Teaching a Real-World Software Design Approach Within an Academic Environment

Jeanne L. Murtagh, John A. Hamilton, Jr.
Air Force Institute of Technology / Joint Forces Program Office

Abstract

In this paper, we discuss how object-orientation and the industrial standard for software development, "Software Lifecycle Processes, IEEE/EIA 12207.0-1996" can be used to enhance the students' design experience in a 400-level course in a software engineering program. Although every phase of the software development lifecycle is important, we have found that the two lifecycle phases which present the greatest challenges in a classroom environment are the software requirements analysis phase and the software design phase. Practical experience with requirements definition is an important part of the software engineering process, but it is beyond the scope of this paper; we shall focus on the latter phase. Furthermore, we claim that students benefit from first developing proficiency in the software design phase -- before they face the major challenges of the software requirements analysis phase. This design proficiency can be developed by employing the industrial evaluation criteria, described in IEEE/EIA 12207, for both the high-level and detailed designs produced by the students. We also recommend that the students work from clear, consistent, and reasonably complete requirements provided by the instructor in the project assignment. It is critical that the students continue beyond the design phase and actually implement, test and document their projects, because some design flaws are not obvious to inexperienced software developers until they attempt to write code based on the flawed design.

This paper discusses how we focused on the design phase of software development and encouraged design practices that would be effective for industrial projects, as well as our much smaller academic projects.

I. Introduction

In this paper, we discuss how to focus on teaching software design principles that can be applied on industrial-sized projects, within the constraints of an academic environment. We found that both an object-oriented approach to software development and extensive use of the industrial standard for software development, IEEE/EIA 12207, "Software Lifecycle Processes," made it possible to provide "industrial strength" design experiences within the size and time constraints imposed by an academic environment. The IEEE/EIA 12207 design evaluation criteria are

pedagogically pragmatic and provide a practical means to introduce students to professional standards of practice.

Before we discuss our implementation of this design-focused class, it is important to understand where the class fits in a software engineering curriculum. We believe that this approach is also applicable to other engineering disciplines. We need to look at the prerequisites for this class; we must also consider what the desired student outcomes are from this class and how the students will be expected to use that information in subsequent classes. This will explain why we chose to focus this class on software design, based on mature requirements provided by the instructor in the project assignment, rather than asking students to conduct an extensive software requirements analysis and then to design their software to the requirements that the students themselves developed.

The body of this paper discusses the prerequisites for and desired student learning outcomes of this class, our rationale for providing solid requirements in the project assignment, how an object-oriented software development approach helps ensure that classroom design experience can be applied to industrial-sized projects, and how students and instructors can use the industrial standard, IEEE/EIA 12207, to aid in the development and evaluation of both high-level and detailed designs. We also discuss the advantages of this approach, as well as its disadvantages and several ways to mitigate these disadvantages.

II. Prerequisites

Prerequisites for this design-focused class include an understanding of fundamental computer science skills and constructs. Students must already understand the three basic control constructs: sequence, selection and repetition, in addition to both basic data types (those provided by the programming language) and data structures (e.g., linked lists, trees, queues and stacks). Our students had extensive exposure to the concept of abstraction, starting from their first computer course. The students had also had a preliminary introduction to the more complicated object-oriented concepts of inheritance and polymorphism. Therefore, students enrolling in this design-focused class had mastered the fundamentals of computing, and were ready to use these concepts in the design of much more complicated programs.

III. Desired Student Learning Outcomes

There were several desired student learning outcomes for this design-focused class:

1) Students shall make use of knowledge acquired in their previous computer science courses to develop moderately complex computer projects, based on clear, consistent and reasonably complete requirements provided by the instructor in the project assignment.

2) Students shall develop high-level (architectural) designs for their software and its interfaces, and get instructor approval before proceeding to detailed design. Students shall demonstrate that all requirements from the project assignment have been allocated to some portion of the high-level design.

3) Students shall develop a detailed design for the software and all interfaces, and get instructor approval before proceeding to implementation (i.e., writing code).

4) Students shall develop a test plan for each project. The test plan shall demonstrate how all software requirements are to be tested. The test plan shall also include a schedule for software testing.

5) Students shall develop documentation showing how their software designs and test plan conform to the requirements appearing in IEEE/EIA 12207.

Note that it was critical for the students to implement and test their projects, even though the focus of this class was primarily on software design and the use of advanced object-oriented features. This is because some design flaws do not become obvious until one attempts to develop code based on the flawed design.

IV. Solid Requirements

In order to give the students a solid design experience in this class, we felt it was critical to provide clear, consistent, and reasonably complete requirements to them in the project assignment.

Our rationale for this is that the requirements form the basis, or foundation, for the entire design. If you want students to focus on developing sound design principles, you must provide them with reasonable requirements as a starting point. If you fail to do so, many of your students may get inadequate design practice, simply because they did a poor job on requirements analysis. Students who are working from their own incomplete or inconsistent requirements will not learn as much about how to translate those requirements into their high-level designs, their test plan and their detailed designs as students who started with a reasonably mature set of requirements.

Note that we are not claiming that requirements analysis is unimportant; we are simply suggesting that students who already understand how to develop designs for moderately complex projects -- based on solid requirements -- are better prepared to perform requirements analysis in the software engineering classes following this design-focused class. This is true because these students are more likely to recognize unclear, inconsistent or incomplete requirements as they consider how those requirements might flow into a high-level design.

Let us consider the analogy between teaching someone to write an article and teaching someone to write a computer program, in order to provide further justification for the importance of having developed good design skills before one attempts extensive requirements analysis. [1]

A suitable foundation must be laid while teaching writing skills. One cannot write good articles without being able to write a good paragraph; one cannot write good paragraphs without being able to write good sentences; one cannot write good sentences without understanding how to use the individual words. You must lay the foundation for good writing by starting at the bottom, ensuring that the words are understood, and then build upward toward the more complicated structures of sentences, paragraphs and then entire articles.

Teaching someone to develop software is similar to teaching someone to write. We often *develop* software "top down;" or at least by the "sandwich approach" of top-down requirements analysis and bottom-up analysis of reusable components available to us; however, that does not

necessarily mean that we should *teach* software engineering in that order. Good software engineering, like good writing, depends on the development of a solid foundation of knowledge before complicated feats are attempted. We cannot efficiently develop software if we do not have a solid understanding of the capabilities and limitations of the computer. This is something we must learn about from the "bottom up," starting by writing very simple programs. After this, students can learn about low-level design (e.g., how to build a single module, or a simple program consisting of several very simple modules). Next, they learn about data structures. These topics are all covered during the prerequisites to this design-focused class. When students have completed the prerequisite courses and understand all of these building blocks, the students are ready to learn to employ these building blocks in more complicated designs. That is the focus for this class. After gaining a solid understanding of the design process, students are finally ready to move on to the critical step of analyzing requirements.

Therefore, we propose that, in order to maximize attention paid to software design within the time constraints of an academic setting, it is advantageous to provide the students with a mature set of requirements. This mature set of requirements can also serve as a good example -- to be recalled and used for comparison on later projects when the students must accomplish the requirements step themselves.

V. Object-Oriented (OO) Design Approach

Students were required to use an object-oriented approach to software development on all projects. This requirement is very helpful in ensuring that design experience gained in the classroom is also applicable on industrial-sized projects, which are typically very large and unavoidably complex. [2] This OO approach, coupled with submission and evaluation of first high-level designs and subsequently detailed designs, forced students to employ design approaches which could be "scaled up" for use on industrial-sized projects, even though our academic projects were very small in comparison. An OO approach supports -- and even enforces -- a number of critical design principles, many of which are equally applicable to engineering fields outside software development. We shall discuss several of these principles below. However, you should note that this is not intended to be a complete list of the design principles supported by an OO approach to software development.

Abstraction is probably the single most important design concept, because proper use of abstraction helps manage the complexity which is inherent in many large design efforts. An OO approach forces students to develop their systems based on objects. Students can focus on the correct level of detail for the current design stage. Initially, students simply consider what objects will be present in their design, and which objects must interact. As the design progresses, students can add detail to the definition of each object. This typically includes attributes (i.e., data) and operations that can be performed on or by each object. Additional details, such as the specific data type for each attribute, or the particular signature (i.e., parameter profile) for each operation, are provided as the design continues to progress from a high level of abstraction to much lower levels, until the design contains enough detail to permit coding.

Modularity is another key design principle that is well supported by an OO approach. Like abstraction, this design principle is also important for engineering areas outside software

development. Modularity requires that the components of the system be designed so that they are highly cohesive and loosely coupled. Interfaces between components must be "clean," meaning that information which will be passed between components must be clearly identified and the amount of information passed over the interface should be minimized. [3] The OO concepts of encapsulation (i.e., packaging related things together) and information hiding further enhance a system's modularity. At a high level of design, modularity supports abstraction. At lower design levels, it has practical application within an industrial setting, by partitioning software into work units which can be accomplished by different people. [4]

The last important design principle we'll discuss is reuse. Reuse helps prevent us from "reinventing the wheel," allowing us to use designs and code -- which have already been developed, implemented and tested -- again in subsequent projects. This has both technical and business (i.e., financial) benefits. [5] The advanced OO constructs of inheritance and polymorphism enhance the likelihood that software can be reused.

VI. High-Level Design

IEEE/EIA 12207 describes the industrial requirements for high-level, or software architectural, design in its paragraph 5.3.5. Both instructors and students can benefit from the use of this industrial standard within an academic environment. The standard describes what must be included in a high-level design, and emphasizes that it is critical to ensure that all requirements are allocated to the software components developed during this design stage. The standard also emphasizes that high-level design includes more than just determining what software components will be required for a given system. Top-level interfaces must also be defined now. Furthermore, the standard requires that preliminary test requirements and a preliminary test schedule be considered at this time. The standard includes the following six criteria for use in evaluating software high-level designs: [6]

a) Traceability to the requirements of the software item
b) External consistency with the requirements of the software item
c) Internal consistency between the components of the software architecture
d) Appropriateness of design methods and standards used
e) Feasability of detailed design
f) Feasibility of operation and maintenance

In order to attain maximum benefit from use of this standard, students must do more than simply follow the standard during their project development, and submit documentation describing their software architectural design at the project's final due date. Instructors must evaluate each student's high-level design, in accordance with the criteria in IEEE/EIA 12207, before students are permitted to proceed to detailed design. Furthermore, instructors must evaluate each student's final submission for conformity with -- and the rationale for deviations from -- this high-level design. This emphasizes the importance of architectural design.

VII. Detailed Design

IEEE/EIA 12207 describes the industrial requirements for detailed design in its paragraph 5.3.6. The importance of ensuring that all software requirements are, indeed, included in the software

design is reemphasized in this paragraph. The standard requires that the software's design be refined into lower levels containing software units that can be coded. The requirement to develop detailed designs for the software's interfaces, including those external to the software item (e.g., user interface), those between the software components and even those between the software units, is especially well stated in this standard: "The detailed design of the interfaces shall permit coding without the need for further information." [6] This helps answer the classic engineering question, "When is the design phase complete?" The criteria used for evaluation of detailed designs are quite similar to the criteria used for high-level designs: [6]

a) Traceability to the requirements of the software item
b) External consistency with the architectural design
c) Internal consistency between the software components and software units
d) Appropriateness of design methods and standards used
e) Feasability of testing
f) Feasibility of operation and maintenance

These detailed designs must be treated the same way the high-level designs were, in order to ensure the maximum benefits from use of this standard during this design phase. Instructors must evaluate each student's detailed design, in accordance with the criteria in IEEE/EIA 12207, before students are permitted to start coding. As with the high-level design, it is also necessary for instructors to evaluate each student's final submission for conformity with -- and the rationale for deviations from -- this detailed design.

VIII. Advantages of This Approach

There are a number of benefits to using this industrial standard in a classroom.

The ABET evaluation criteria state that "the scope of the design experience within a program should match the requirements of practice within that discipline." There can be no better way to accomplish that than to have the students read and follow the standard that guides industrial practices!

Requiring students to submit their architectural designs -- and get instructor approval before proceeding to detailed design -- gives instructors a chance to catch and correct errors in students' designs before those errors are propagated throughout the entire project (i.e., to lower level designs, to code, to testing and to documentation). This prevents students from investing a significant amount of time on a fundamentally flawed approach. Evaluating the students' detailed designs before they proceed to coding provides a similar, although slightly less critical, benefit. Although both of these design reviews do take instructor time early in the project cycle, the reviews may actually save time overall. Instructors will see fewer panic-stricken students asking for help as they frantically try to get running code from a fundamentally flawed design during the last week (or perhaps even the last few days) of the project cycle.

The design submission requirement has the additional benefit of supporting "writing across the curriculum" initiatives. Students gain practice in technical writing, which is essential to their ability to function effectively in industrial positions. This also reinforces the idea that documentation is a critical part of the software development lifecycle, rather than something to

be accomplished hastily after the "real work" (which, as viewed by the students, is sometimes simply writing the code) is done. Recall, too, that IEEE/EIA 12207 requires test planning information at both the architectural and detailed design levels. This reinforces the importance of test planning throughout the project. Contrast this with an all-too-common student approach: "It compiled; it ran; I have output; it's ready to submit! What do you mean, how do I know if it's working correctly? I told you: I have output! Oh, you want to know whether it's the output I should expect. Gee, I didn't have to time check ***that***!"

Note that it is critical to grade the students' submissions, rather than simply providing an ungraded "review" of the students' design work. Furthermore, in order to maximize the benefits of these early submissions, the students' final projects must be graded for conformance to -- or the rationale for deviations from -- the original design submissions. These three grades (high-level design, detailed design, and conformity of final project to original design) force students to develop some practices that will be necessary in industry but which students frequently attempt to neglect in an academic environment. We recommend grading documentation on both technical content and grammatical correctness!

One of the most common student tendencies is procrastination. A truly dedicated procrastinator is not defeated when an instructor simply requires students to submit "review" copies of high-level and detailed designs early in the project development cycle. Without grading for motivation, the dedicated procrastinator is likely to use this approach: "I'll take a really quick look at the project assignment and write up something -- ***anything*** -- to turn in now; I'll worry about the ***real*** design later (perhaps a few days before the project is due)." To defeat this type of procrastinator, the instructor must grade the high-level design and the detailed design. The instructor must also grade the final project submission for conformity to the original design. Software development is often an iterative process, so complete conformity to the original design might not be possible, or even desirable. However, grading the final project for conformity to the original design can provide the students with additional motivation to be as thorough as possible as they develop their original designs. Requiring students to explain the rationale for deviation from their original design provides additional practice with technical writing, and gives students a realistic view of the importance of documenting the inevitable changes they will make when they are employed constructing industrial software projects.

IX. Disadvantages of This Approach; Ways to Mitigate Them

The primary disadvantage of this approach is the requirement for fast grading turn-around times on the students' high-level and detailed designs. The instructor must also spend some time grading the final project submission for conformity to the original designs; however, we feel that this is time well spent, and this paper does not discuss ways to minimize this time.

There are a number of ways to mitigate the fast grading turn-around time. The instructor must select the design submission dates with care. It might be helpful to receive the design submissions on a Friday, so the instructor will have a few extra days before designs must be returned to the students. It is also possible to interleave reading assignments between design submission and the return of the graded design, so students will be able to use their "homework/project" time productively until the instructor can return the graded designs.

If it is financially feasible, it might be wise to employ more teaching assistants (or graders) for this course than your department would normally assign. This might seem expensive initially. However, if your students develop good design habits in this class, it may actually be easier to grade projects in later classes.

If it is impossible for the instructor to provide rapid turn-around times for all students, a selective review process should be considered. Ensure that all students understand that the instructor is available to review their designs and answer questions. Require only students who did poorly on the previous project to formally submit their designs early in the development cycle for this project. We have used this approach with great success.

X. Conclusion

This paper has discussed how an object-oriented approach to software development and extensive use of the IEEE/EIA 12207 standard can help an instructor focus on the design phase of software development and encourage design practices that would be effective for industrial projects, as well as much smaller academic projects.

We have discussed the prerequisites for and desired student learning outcomes of this class, our rationale for providing solid requirements in the project assignment, how an object-oriented software development approach helps ensure that classroom design experience can be applied to industrial-sized projects, and how students and instructors can use the industrial standard, IEEE/EIA 12207, to aid in the development and evaluation of both high-level and detailed designs. We also discussed the advantages of this approach, as well as its disadvantages and several ways to mitigate these disadvantages.

Bibliography

1. Murtagh, J.L., "The Software Development Process: An Integral Part of Weapon Systems," *DSMC Program Manager Magazine*, Nov-Dec 1992.
2. Brooks, F.P., Jr. *The Mythical Man-Month: 20th Anniversary Edition*, Addison-Wesley, Reading, MA. 1995, p. 182.
3. Ibid. p. 220.
4. Ben-Ari, M., *Ada for Software Engineers*, John Wiley & Sons, Chichester, England. 1998. pp. 3-4.
5. Ambler, S.W., *The Object Primer: The Application Developer's Guide to Object-Orientation*, SIGS Books, New York, NY. 1995. pp. 6-7.
6. IEEE/EIA 12207.0-1996, "*Software Lifecycle Processes*," Institute of Electrical and Electronics Engineers, Inc., New York, NY. 1998. pp. 18-19.

JEANNE L. MURTAGH
Jeanne Murtagh is an officer in the U.S. Air Force. She is currently the director of the Software Professional Development Program (SPDP) at the Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio. Her previous assignment was as an Assistant Professor in the Department of Electrical Engineering and Computer Science at the U.S. Military Academy at West Point, NY.

JOHN A. "DREW" HAMILTON, JR.
Drew Hamilton is an officer in the U.S. Army. He is currently the director of the Joint Forces Program Office in San Diego, California. He is also an adjunct faculty member of the Naval Postgraduate School. He has previously served as the Research Director and an Assistant Professor in the Department of Electrical Engineering and Computer Science at the U.S. Military Academy at West Point, NY.