disciplinary Commons

# SOFTWARE ENGINEERING PORTFOLIO

- Gary R. Mayer, PhD

## INTRODUCTION

This document provides one educator's portfolio for Software Engineering instruction. It has been developed while participating in a Software Engineering Disciplinary Commons group. The group itself is structured by the definition of a Disciplinary Commons group with the intent of promoting education in the Software Engineering field through the sharing of ideas. The group's work may be found at http://sec.cs.siue.edu/.

### Purpose

The purpose of my portfolio is to provide an overview of the Software Engineering course that I have now taught twice and will be teaching again in the future. As I am a relatively new professor (I began instructing in the Fall 2009 semester), it also includes my reflections of what I believe has worked and what has not.

I believe that the reader is best served by reading this document and reflecting on his or her own educational environment. It is unlikely that you will find this document well-suited as a template for your own course. Rather, I think that you may find parallels in situations and, perhaps, things not thought of before. Then, this document may enable you to draw ideas or give you that extra nudge in the right direction in order to improve your own educational approach.

### Audience

The portfolio is intended for any instructor who is currently teaching, or plans to teach, a software engineering course. While it is primarily aimed at the Software Engineering instructor, others may find some ideas more generally applicable.

## My Background

Serving as a professor at a university is my second career. I earned my doctoral degree in Computer Science from Arizona State University in 2009. My dissertation summarizes my research in formal modeling and simulation. My Master's of Computer Science was received from Southern Illinois University Edwardsville in 2004. My thesis captured my research in using robotics in education. My undergraduate degree is a Bachelor's of Science in Mechanical Engineering with Aerospace interest, received from Worcester Polytechnic Institute in 1992. My senior project (called at Major Qualifying Project at WPI) involved calibration of pressure transducers and thermocouples in a converging-diverging nozzle capable of achieving mach flow. My Minor Qualifying Project (done in my junior year) consisted of devising physics laboratories for a local high school class using equipment that supported their minimalistic budget.

Upon completion of my Bachelor's degree, I was commissioned a Second Lieutenant in the United States Air Force. During the almost ten years of my active duty service, I performed as both a program manager and an aeronautical engineer. I have worked with teams of contractors, civilians, and military personnel. I have stood up projects with multi-million dollar budgets that are still operating today and shut down weapon systems that had operated longer than I had been alive. My work has exposed me to embedded systems and globally deployed systems.

I mention all of the above because it shapes my beliefs about what students need to know from a software engineering course, and about what they should get from a college education overall. I think that you should know so that you can better decide what's appropriate for your course.

## TEACHING PHILOSOPHY

I believe that the purpose of university instruction is two-fold – provide students with *knowledge* and techniques for *knowledge discovery*. A student who understands the basic concepts when faced with a similar problem will be more apt to recall the information needed to develop a solution. However, it is often the case in real life that the problem at hand does not fit the known problem-solution sets. Thus, while an instructional approach may facilitate learning new material in the classroom, an instructor must also provide students with an understanding of how to advance their knowledge beyond the classroom.

While the material to be taught comprises the foundation of the instruction, and my goals of providing knowledge and discovery techniques create the frame; my method of delivery is tailored to fit the students' needs. I use factors such as class size, undergraduate versus graduate curriculum, and student background information such as educational and career experiences, to help shape my lesson plans. This enables me to devise explanations and sample problems to which the students can relate and understand. These factors also direct my selection of instructional approaches. Group projects, in-class discussions, daily quizzes, and student presentations can all be effective tools, but only if employed within an environment that is suitable for the approach.

At the same time that I am teaching the material, I teach the students how to further educate themselves. One knowledge discovery approach that I use is to lead the students through one or more exercises of how think about a problem, in general terms, without focusing on the final implementation. I discuss the problem, starting from a large concept, and then decompose it into smaller pieces. I also discuss taking known, simple ideas, and using them to create a more complex system. I highlight the assumptions that I have made and discuss the implications of relaxing those assumptions. For introductory courses, the discussions also include the creation of

a programming language-free algorithm that describes a software approach to a solution. Finally, I provide the students an environment in which they can explore these concepts on their own.

Example problems, projects, and culminating projects guide students toward creative thinking. The example problem sets that I assign are often bound to simplistic cases in order to make a new concept easier to understand. I then assign projects that require the students to bridge a small gap between the instruction and the project solution. These projects are a core piece of my lesson plan and are designed to allow the students to demonstrate their understanding of the subject matter in a manner that makes sense to them. Thus, when they make errors in their solution approach, I can then discuss these in terms with which the student can relate. This, in turn leads to better comprehension of the material and its applications. Furthermore, it challenges the students to think about why a concept works or does not, vice just mechanically repeating words on an exam. By encouraging students to abstract problems and discover solutions, I hope to create individuals who will perform better outside of the classroom, both within the workforce and the research laboratory. The culminating projects offers the students an opportunity to apply many of the new concepts to a more complex system, correcting previous mistakes and gaining a better understanding of how the concepts apply outside of a micro-problem set.

The above is obviously filled with good intent, but more easily said than implemented. I think that overcoming the challenge of implementation lies in an understanding of today's college student. It is my impression that many college students, while somewhat interested in the topic at hand, don't quite have the enthusiasm for learning the material. Is this due to a lack of critical thinking and self-motivation stemming from high school curriculum? Perhaps the family *encouraged* the student to major in the field for monetary reasons and that is the only reason for being there. Could it have just been something the student knew about on the periphery but knew none of the details and associated work involved? Is the Master's student there to increase knowledge, or just to get the paper entitling one to higher pay or a longer stay in the United States? Knowing

the answers could reshape a class; obtaining the answers and acting upon them would require getting to know every student on a personal level and an inordinate amount of time devising educational materials tailored to each student. Reality dictates that this is not possible with given resources (especially time). So, I strive to devise a course that challenges those who demonstrate self-motivation and a desire to learn the material. I use them as my measuring stick to determine how well I am providing the material. The others are neither forgotten nor perceived as unimportant; I strive to ensure the basics are understood by the majority and that all of my students have a foundation from which to broaden the understanding with which they depart my classes. However, education requires effort on the parts of both instructor and student.

# COURSE CONTEXT

## Background

Southern Illinois University Edwardsville (SIUE) offers two computer science degree programs – a Bachelor of Science and a Bachelor of Arts. The degree programs are Accreditation Board for Engineering and Technology (ABET) accredited. CS 325, Software Engineering, is considered a "core course" for both programs. As such, it meets some of the key elements stressed as part of the ABET accreditation. In both curriculums, Introduction to Computing III[1] is a prerequisite to CS 325, which in-turn is a prerequisite to the senior project courses, CS 425 and CS 499. The students who take CS 325 are mainly junior and senior computer science majors, with some sophomores in attendance. Many students live locally (on campus and off), and many have jobs.

Current constraints for this course include the fact that it is offered every semester (as are all of the department's core courses) and that it currently employs four textbooks. The course covers a broad array of topics, most of which the students are seeing for the first time. Specifically, the course covers software engineering as a discipline – from requirements analysis to support and maintenance. It provides an introduction and experience with the Personal Software Process (PSP). PSP is used as a means for the student to assess his/her own performance and to gather data for estimation purposes on the final project. The course also provides an introduction to UML. Finally, it provides material on lessons learned in software engineering. This course is the only course the students will take that provides a software engineering overview and, in most cases, an in-depth discussion of UML prior to senior projects (CS 425 and CS 499).

The four textbooks that the course employs are:

[1] The course is identified as CS240 and provides basic software engineering concepts, elementary data structures and algorithms, and fundamentals of object-oriented programming.

- Brooks, F.P., Jr. *The Mythical Man-Month: Essays on Software Engineering*, anniversary ed., 1995.
- Humphrey, W.S. *PSP: A Self-Improvement Process for Software Engineers*, 2005.
- Miles, R. and Hamilton, K. *Learning UML 2.0*, 2006.
- Tsui, F. and Karam, O. *Essentials of Software Engineering*, 2007.

The book by Tsui and Karam provides the core of the course material. The other books provide more in-depth information on specific topics than what the Tsui and Karam book does. However, the fact that the course employs four textbooks makes it a challenge to integrate the information from the texts. The presentation of similar topics is provided in a different order, with a different focus, and sometimes with different information. Also, given the breadth of knowledge provided by the course and necessary for the students to be prepared for senior projects, there is a lot of work for both preparation and grading. As the course is offered every semester, it does not provide a lot of time to reflect on the previous semester in order to plan changes for the next semester.

To the students, the use of four books offers its own challenges. Financially it is not a burden as the university uses a textbook rental system for undergraduate students. However, the students in my two previous courses – including those that I would consider to be making an extra effort to learn the material – have remarked that it is difficult to keep up with the reading given the large volume of material across so many books. Also, it is difficult to study for exams, again due to the volume of material across multiple books.

Until recently, the course also provided an introduction to design patterns using Shalloway and Trott's, "*Design Patterns Explained: A New Perspective on Object-Oriented Design"*, 2 ed., 2005. However, it was conceded by the three instructors who taught the course (myself included) that the students, having just learned UML and having just had their strongest exposure to object-oriented programming, could not quite

develop the appreciation for and understanding of design patterns as an experienced programmer might. Furthermore, the reduced content allowed a stronger focus of the UML material that we felt was more important. It is very infrequent, but special topics courses such as design patterns have been offered in the past.

The course will continue to evolve as additional changes are made to the overall curriculum. At the start of the Fall 2010 semester, the department faculty agreed to transition from using C as the introductory programming language to Java. Along with this change comes an introduction to UML in the form of class diagrams and, in the next course, use case diagrams and perhaps exposure to sequence diagrams. Once these changes are implemented, Software Engineering will again be revised. It is anticipated that there will be a stronger emphasis on sequence diagrams and other diagrams such as component diagrams. Additionally, the reintroduction of design patterns may be reconsidered.

## Approach

In keeping with my teaching philosophy, I try to provide lectures that summarize key elements of the material and attempt to allow the students to explore aspects of the material on their own during in-class exercises and homework. The lecture material provides the immediate knowledge, which I support through examples that may be real-world experiences and/or relate to experiences and concepts that the students are likely to have had during their academic careers. The in-class exercises have been conducted in groups of 3 – 5 students. I have noticed that while it is often difficult for me to obtain responses from the class when they are seated individually, they are quite open to discussing ideas and posing questions when they feel supported by their peers. The in-class exercises and homework problems are intended to have the students think about how to apply one or more recently learned concepts. Initial feedback garnered from the classes indicates that they feel this has been a good usage of class time.

I was troubled; however, by the fact that a periodic quiz given in class, even if provided right after lecture material was covered, often yielded poor results. I often devise quizzes such that there are matching questions, fill-in-the-blank, and some short answer. A few of these questions are based off of diagrams that come direct from the lectures and reading material. I have tried administering quizzes both right after the lecture that covered the material and the next lecture, to provide students time to re-examine material they may not have fully understood and/or ask me questions. The Mid-term Exam utilized the same format and the results were grades with an average percentile of 69 and a median percentile of 69.5. The high score was 86. Modeling the format after one used by a fellow Disciplinary Commoner, I have since changed the format of quizzes to open book problems.

I now provide my quizzes to the students at the start of class and provide a minute or two for them to look through it before beginning the lecture. All of the answers are contained within the lecture I provide. In this manner, I hope to keep the students interest as they are mentally engaged with the lecture to some level. This has also promoted some discussion at the end of lectures if a student did not understand the quiz question or material. A downside of this approach was reported by the students who frequently took notes. They stated that they felt they did not have enough time to adequately take notes and complete the quiz that is due at the end of lecture. To compensate, I began allowing five minutes or so for the students to complete the quiz, transferring information from notes to the quiz sheet. In my second Software Engineering course, the Mid-term exam, which was very similar to the first course exam, had an average percentile of 77.1 and a median percentile of 79. While it is difficult to claim statistical significance based upon the scores alone, student feedback was more positive and student application of the material to projects was also improved.

The Final Exams for my Software Engineering courses are open book. While I want to test the students' capability to recall and understand key concepts in the Mid-term, in the Final Exam I am more concerned with assessing their ability to apply what has been learned. As such, I allow them to bring all course-assigned

textbooks, one additional textbook of their choosing, and a standard notebook page of hand-written notes (the typical “crib” sheet). I allow an alternative textbook because, despite our best efforts as instructors to choose a good textbook, there are always some students who will find the material better explained in another. The notes sheet allows the students to pull key concepts from the lecture slides. The fact that the notes sheets are hand-written provides two purposes. First, the student must spend some time thinking about the material being written. Second, I collect the notes sheets with the exam and this provides me feedback as to what the students felt was important enough to actually spend time writing.

While I find my exam approach to be best suited to the material and my approach to teaching, it does require a lot of work both in production and grading. A typical question may provide requirements and request that the students draw a use case diagram. Another question will provide a use case diagram, use case descriptions, and class diagrams, and request that the students draw sequence diagrams. The questions are both time consuming to construct and time consuming for the students to answer during an exam. Thus, I have spent a lot of time devising questions that emphasize important points, only to have to remove one or two in order to accommodate the time commitment required by more important questions. Furthermore, there is rarely one “correct” answer. Therefore, these types of questions require that the grader understand the material well and to follow a student’s train of thought in order to determine the amount of credit to be given.

## COURSE CONTENT

The course covers a broad array of topics, most of which the students are seeing for the first time. Specifically, the course covers software engineering as a discipline – from requirements analysis to support and maintenance. It provides an introduction and experience with the Personal Software Process (PSP). The course also provides an introduction to UML and (when I first taught it) to some of the basic design patterns. Finally, it provides material on lessons learned in software engineering. The selection of topics, and the books to present them, was pre-determined by two faculty members who created the course based upon their previous experiences with same. The software engineering course is intended to fill gaps in the student's education in order to prepare them for their senior projects and the "real world". The order of the topics was left to me.

The first time that I taught the course, I began within an overview of systems and systems engineering, including various approaches. The Personal Software Project was also introduced. We discussed requirements analysis and then design. At that point, I incorporated the lessons on UML – which focus mainly on design – and related Use Case diagrams and description back to requirements analysis. I focused on key UML diagrams but tried to allocate time for all of the diagrams. From UML, it seemed a logical progression to begin discussing object-oriented design patterns. Again, I tried to allocate time for all of the patterns. Other course topics will include support and maintenance and lessons learned from other software engineering projects. I believe that this ordering lends itself well to the typical system development life-cycle.

The students' grades are project focused. Twenty-five percent of their grade is from individual projects, and another twenty-five from group projects. Homework and quizzes are ten percent each, and I drop the lowest score from each category when determining these category portions of the final grade. There is a Mid-term Exam and a Final Exam. Each exam accounts for ten percent of the total grade. I believe that software engineering is an applied set of knowledge and skills and, as such, is best assessed in that manner.

Earlier individual projects are primarily focused on familiarizing the students with the Personal Software Process (PSP) (see IP #2). In latter individual projects (*e.g.*, IP #4), more course-specific topics are included along with the PSP element. For example, IP #4 attempts to concretize UML use case diagrams and description, as well as class and sequence diagrams, by having students code to a pre-specified design and accompanying requirements analysis. The intent was to help them take the next step from having a UML design to being able to implement it. By providing the design, I reduced the risk of the students trying to code to errors within their own designs.

The homework in the course has been used to try and reinforce on or more topics recently covered in class. Some are very stand alone. Others have been the requirements analysis portion of an individual project. Similarly, the quizzes are intended to motivate a student to remain current with the reading and to try and reinforce some of the topics taught during lectures. While I believe that I have seen some partial success with the homework, the same cannot be said for the quizzes. If nothing else, the quizzes have demonstrated that, often, the students are not coming to class prepared and/or are not readily absorbing the material provided during lectures. If nothing else, they provided the students with some indication of how the Mid-Term exam might look. The in-class exercises seem to be a better method of reinforcement learning. However, while group projects assist the students in their understanding, it becomes increasingly more difficult to individually assess a student's understanding of specific material.

As stated in the Course Context section, the Mid-term results were not spectacular. They were not horrid but obviously showed that many students had neither recall nor recognition of many of the elements provided in the first half of the course. Many students remarked that there were too many similar terms and too much material distributed across four of the five course books that were used in the first half of the course semester. As I intend the focus of the second half of the course to be on UML and design patterns, I have revised quizzes to be open book problems. Similarly, the Final Exam is planned to be open book with the focus less on describing a

specific term and more on demonstrating an understanding of a bigger picture in the application of UML and design patterns. This may work better in that I found creation of the Mid-term to be difficult in that I had to cut a number of problems that focused on specific areas that I thought relevant to test. However, the large volume of information spread across so many books made the exam too lengthy. As such, a number of problems were cut from the exam.

The Final Exam, as planned, was open book. The students were allowed to bring all five course textbooks, one additional textbook of their choice, and a double-sided, 8 ½" x 11" sheet of notes. The Final Exam focused on UML and design patterns, but also included some material from the end of the Tsui and Karam book – specifically, testing and configuration management. The average of the Final Exam was worse than the Mid-Term, with only one-quarter of the class receiving what would be considered a passing grade and the high-score was a 76%. While there was a lot of material on the exam, this is not the reason for the students' weak scores. The low scores were not due to the lack of time in answering questions; rather in the inability to effectively devise many UML diagrams (*e.g.*, a sequence diagram) and an inability to recognize the best design pattern for specific situations. Furthermore, I do not feel that the exam was a poor assessment of student knowledge. I had interacted closely with most of the students over the semester and those who gave me the impression that they had the strongest understanding of the material were the same ones who passed the exam.

What I point to as the failure in this case was my continuing on to provide an introduction to design patterns when the quizzes indicated that the class did not yet have a full grasp of UML. I had structured the material based upon my predecessors and gave equal time to UML and design pattern material. The problem is, this is the first time that the students have seen either. From my own experiences, design patterns are not something one just grasps, even if the fundamentals of UML are well known. I believe design patterns are best understood once the individual has had an opportunity to apply UML in a myriad of ways, to get stuck with the basics, and to get to the point where the individual is almost devising their own patterns. At this point, when design patterns

are introduced, the need and the benefit of such artifacts are better understood and, during the exploration of plausible solutions, UML itself is no longer such a new thing. Therefore, I revised my approach in the next software engineering course that I taught in order to provide a better exposure to UML. Design patterns were discussed, but the breadth to which they were taught was reduced.

The second time that I taught Software Engineering, I began by discussing software systems, PSP, and software engineering as a discipline. I then introduced specific UML diagrams with the various software process elements to which they best correspond. So, for example, after discussing requirements engineering I presented use case diagrams. After system design, we discussed the details of class diagrams and sequence diagrams. Providing the UML material earlier in the course and with other material in between allowed me to integrate more UML into assignments. It also appears that breaking up the introduction of diagrams allowed the students time to better comprehend the material before moving on to the next.

# INSTRUCTIONAL DESIGN

This section contains my musings on creating a course.

## Overview

It's hard to say why particular topics are taught in specific ways. Often times, I believe it is the experiences that the educator has had. S/he learned a topic a particular way or has had first-hand experiences of a particular nature, and it is this context in which the instructor views the content of the course and, thus, devises methods of delivery. I believe progress is often made incrementally s an instructor will work from this foundation and begin to step away from it in order to bring in new approaches that may not be as familiar or are completely unfamiliar.

I think that one could argue that there are an infinitesimal number of ways that content can be taught. There is no best as good instruction depends upon three factors – the instructor, the material being taught, and the student. While, in most cases, one might say "it is best for most students if Subject X is taught using this approach," it is very possible that the instructor just can't adequately implement the approach. Therefore, though another approach may be typically less suitable approach for Subject X to most students, given a particular instructor it may be the best alternative.

There is sometimes "leaps" in one's instructional design. These might be personal "Eureka!" moments or just having seen/read/heard about a significantly different approach that an instructor sees as a worthwhile investment for potential gains if the current approach is seen as faulty. Often, textbooks have the same limitations. Textbook versions on subjects often slowly roll the material and the delivery – not only between versions of the same book, but in "competing" textbooks as well. But, sometimes, an "odd duck" comes out. One example is actually a series of odd ducks, the "Head First" series. This series uses a much more light-

hearted, exemplar approach to instruction than most textbooks on subjects such as programming, design patterns, etc. I have heard varied opinions from a few different instructors as to their personal opinion but I have not heard much from students as to the effectiveness of the material presented. Personally, I found the books semi-useful. This may be because I already possess at least some rudimentary knowledge of the material in each of the Head First books that I examined. So, reading through the stories from cover-to-cover didn't feel like an adequate usage of my time and the teaching format the books use don't lend themselves well to topical searches. Thus, perhaps this is an issue arising from my "training" with classical computer science textbooks.

## Course Specific

I see software engineering as being as much of an art as it is a science. While there are definite rules, guidelines, and "best practices" that have been established; when one engineers it is a personal thing based upon the way in which you perceive the system, its environment, and your own. Thus, it is an art and each person is likely to draw different abstractions and, from that, devise a different approach to the same problem as another well-trained engineer. With this in mind, I have devised my software engineering course to first provide the foundation knowledge – those rules, guidelines, and best practices. I try to use the quizzes and homework as motivational factors for the students to commit this knowledge to memory. I then assign in-class problems and individual projects to allow the student to find personal ways to apply this knowledge to exemplar problems. In some cases, I then had the students discuss their results in class. In this manner, they saw how others perceived the same problem and the different approaches that could be taken. Furthermore, when there was a mistake, I was able to discuss it such that all students could benefit from understanding why the approach didn't work and what is a better approach.

As mentioned in the Course Content section, I do not believe the students left the class with as strong an understanding of UML as I would have liked in my first course. As I also stated, I believe this is because I moved

the focus of the material from UML to design patterns before the students fully grasped the most common UML (class diagrams, use cases, and sequence diagrams). I used class group activities, quizzes, individual projects, and group projects as medium by which I could relate the UML material. While student feedback states that the class projects were helpful, most students point to group projects, where they were on their own and had to figure it out, as being the best source of learning. The problem is that an examination of the submitted project material indicates that, while they did indeed learn a lot more about the topic, it was rare that they learned it correctly. Unfortunately, given the deadlines of the group projects, there was little time to correct the students understanding except via feedback that accompanied student grades. Further, given the number of repeat mistakes that were seen in the second group project, I do not believe many students used the feedback while working on the second group project.

In an effort to capitalize on their self-learning, the second Software engineering class that I taught made use of individual projects that built upon one another and a lot of class discussion about each project. With this approach, I had hoped that each student might gain a better understanding of the pros and cons of specific approaches and wanted the student to retain that information for the next individual project and the larger projects at the end of the semester.