A FAILURE-BASED SOFTWARE ENGINEERING PROJECT

Antonio M. Lopez, Jr.
Computer Sciences and Computer Engineering Department
Xavier University of Louisiana
New Orleans, LA 70125
tlopez@xula.edu

ABSTRACT

Software engineering is difficult to do in the real world, so teaching it to computer science undergraduates in an academic setting is a real challenge. Many software engineering instructors, especially those at small, liberal arts colleges or universities, are limited to a one semester course where they seek to use a "real world", term-long, team-developed project to give their students a desired mixture of theory and practice. The project is selected for a variety of reasons – availability of a real client, complexity of the problem, ability of students to have a "running" software product at the end of the term, and others. This paper presents the rationale for a challenging software engineering project that relies upon the instructor's "real world", supports failure as a learning mechanism, and involves the instructor in guiding the team and evaluating each of its individual members.

INTRODUCTION

Teaching a software engineering course is a labor-intensive activity requiring, on the part of the instructor, both technical skills and managerial control in order to provide students with a well-crafted mix of theory and practice. To avoid having the course become simply one of theory (i.e., terms and concepts), instructors have taken a variety of approaches to teaching the subject matter to undergraduate computer science majors. Popular among these is the use of a "real world", term-long, medium-size, and team-developed software project [5]. However, the use of such a project is often problematic in itself.

The first problem is answering the question: Whose "real world" is it? Some educators seek real clients -- people (in- or out-of- house) or organizations (especially nonprofits) -- that need software written for their own purposes [15]. Sometimes educators are approached by people seeking "senior project students" to write software for them and in so doing to receive college credit in a software engineering course [21]. Both these situations raise the ethical question that the instructor must answer: Are people or organizations taking advantage of the students? For this and other reasons, some educators see real customers as unfeasible. Another reason is uncertainty with regard to a customer's commitment to the software development project. Sometimes a real customer begins with a high-level of resolve to get a software product delivered, but within the semester timeframe the customer's commitment to the project migrates to other business concerns. The customer does not return phone calls and/or the customer misses critical meetings with the students. Some educators develop lists of projects ranging from graphically oriented games to web projects with database back-ends, which simulate "real world" conditions [7]. Consequently, the first problem that the instructor must address is the "real world" domain for the project.

The second problem that the instructor must consider is the "goodness of time-fit" that the selected real world conundrum offers to the students. In over twenty-five years of software teaching and consulting experience, the author has never seen a team of 5 ± 2 people having little or no prior experience working together to develop and implement a real world, medium-size, software system in four months -- the

approximate time in an academic semester (i.e., term-long). Inherent in this observation are two facts: (1) Only complex software systems need to be engineered, and (2) Software developers rarely have the luxury of deciding the problem for which they will develop software. Typically in the real world after being assigned a project, the software engineer's first stop is the library where more can be learned about the problem scenario. This takes a great deal of time. So in order to make sufficient progress on a term-long software development project, instructors must have the ability to revise the real world problem, especially if they want students to have some level of success in the allotted amount of time [20]. Time constraints limit the amount of real world conditions in any academic software project worthy of being engineered. So, the second problem that the instructor must face is the instructor's own ability to adjust the project "on-the-fly", if need be due to the experiences of the team and the available time.

The third problem that the instructor must take into consideration is the evaluation of the individuals on the team as well as the team itself. In the first two or three semesters of the undergraduate computer science curriculum, the students are being taught the "programming craft" and being evaluated on their own performance. They are given well-defined, small-sized problem scenarios having software solutions that a single individual can write in a week or two. Such assignments do not require software engineering. Furthermore, in the zeal to get students to learn the programming craft, some instructors allow these simple programming assignments to be turned in late, usually deducting points according to how late it is. Academia allows for numerous individual excuses: "I had a Math test." "My 20-page English paper was due." Granted students are not employees subject to workdays that are focused on meeting a software deliverable deadline. Although, when a student drops the course and leaves the remainder of the team with additional work to do, the instructor can claim a simulated real world death or an employee going to a new job. Nonetheless, most of this is counter to the software engineering culture. Many undergraduates, especially those who tend to be procrastinators or hackers, have difficulty transitioning to a team effort. They have difficulty operating under fixed timelines and meeting deadlines. They do not comprehend sequenced project components (created by others) that are interdependent and necessary for achieving milestones. Hence the instructor must have an evaluation methodology in mind prior to beginning the project.

This paper presents the method the author used to address the aforementioned software engineering project problems. The focus is on the instructor's "real world", a view that failure is a learning mechanism, and the need for active instructor leadership – guiding the student team and evaluating each of the individuals on the team.

THE REAL WORLD

Software engineering has its roots in the Department of Defense (DoD), beginning in 1968 with the coining of the term at a NATO conference and continuing with the establishment of prestigious organizations such as the Software Engineering Institute at Carnegie Mellon University. The defense software community deserves credit for pioneering software process assessment and process improvement technologies [13]. The DoD software engineering work continues today because of the complexity of the problems that safety-critical software systems must solve. Nonetheless, there are only a few ways to increase the probability that the software will operate properly (i.e., to the customer's specifications), but there are thousands of ways to cause a software system to fail.

Pick up one of today's software engineering textbooks [16, 10, 18] and the reader will probably find some level of coverage of the software failure that destroyed the Ariane-5 rocket in 1996. In 1999, the story appeared in a computer science education journal[1] and it was reprinted with permission in yet another

computer science education journal just two years later [2]. Obviously, this was an event worthy of note. The next generation of software engineering texts might highlight the $125 million loss of the NASA's Mars Climate Orbiter in 1999. According to Arthur Stephenson, chairman of the Mars Climate Orbiter Mission Failure Investigation Board [12]: "The 'root cause' of the loss of the spacecraft was the failed translation of English units into metric units in a segment of ground-based, navigation-related mission software … ." Dr. Edward Weiler, NASA's Associate Administrator for Space Science, was quoted as saying [11]: "People sometimes make errors. The problem here was not the error, it was the failure of NASA's systems engineering, and the checks and balances in our process to detect the error." Few would disagree that these are real world examples of failures that software engineering practices were supposed to prevent. So software developers learn from these kinds of failures and produce better practices and procedures that will help them avoid the same kinds of software failures in the future.

The Missile Defense Agency (MDA) has several challenging real world problems that software must solve. Although the idea of "hitting a bullet with a bullet" is both politically and technologically controversial [9], the problem domain is worthy of software engineering. Like the civilian rocket programs previously mentioned, MDA has had its share of failures. The most recent failure occurred in June 2003 and though still under investigation the performance of the solid divert and attitude control system of the interceptor is suspect [8]. Not only must software engineers develop the right software, they must get the software right.

The author is among four faculty members at Xavier University of Louisiana doing research in the missile defense domain. Xavier is a Historically Black College and University, and one of the goals stated in its cooperative agreement with MDA is to encourage African American computer science and computer engineering majors to pursue careers in DoD research and development after they graduate from college. Faculty members are trying to accomplish this goal through involvement of students in undergraduate research [14]. However, since the agreement financially supports only one undergraduate researcher per faculty member, the progress is slow. In Spring 2003, the author was given the software engineering course to teach. To increase the number of undergraduate computer science majors being exposed to the missile defense domain, the author decided to adapt a small problem segment from his real world as the term project for the course. The experiences of the instructor in the problem domain and with the software engineering process set the stage for the context of the project.

As is typical with real world software engineering projects, the instructor began the development process with a rough sketch (See Figure 1) and a vague statement of the problem:

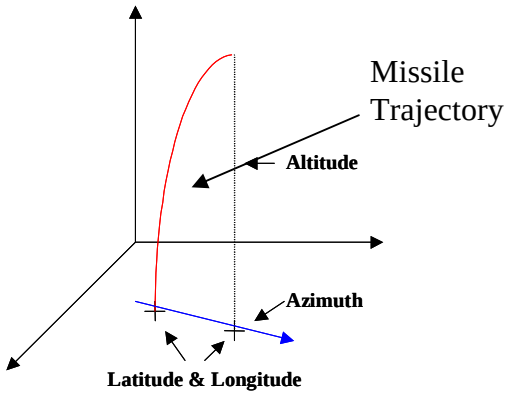

Figure 1. A missile's ascent.

The Problem Scenario The customer wants a software system that will run on their UNIX platform to project and track the trajectory of a missile that has been launched. The launch will be marked when a sensor provides latitude and longitude (lat & long) coordinates of the launch site at time zero. Initially, the software will determine projected lat & long ground position coordinates along with a projected altitude for 20 seconds into the future flight of the missile. Then at 20 second

intervals, sensors will provide the actual lat & long ground position coordinates and altitude of the missile. Before the next interval's data are received the software system will "rethink/correct" its previous projection then calculate and display on the computer screen new projected lat & long coordinates and altitude for the expected future position of the missile. The purpose of the system is to use sensor data to make increasingly better projections as to the where the missile is going. The software system is concerned only with the "boost phase" of the missile; this is approximately the first three minutes of ascent before the missile leaves the earth's atmosphere. All measurements of altitude and distance will be in metric units.

This simple problem scenario was so intimidating that most students were basically in shock. There was no rush to code; the students do not even know what questions to ask of the instructor. Thus, the problem scenario forced the students to focus on requirements and specifications, and to visit the library to find out more about missiles and their trajectories. In this medium-sized, complex software engineering project, there are literally hundreds of questions to be asked, for example: Given just the lat & long of the launch site how can one predict with any kind of certainty the azimuth of the flight? Do all missiles have the same launch velocity? How many types of missiles are there? What countries have what types of missiles? The students were the software developers who had to know enough about the problem domain to form the questions properly. The instructor was the customer who could answer the questions. In the instructor's real world, software engineers spend a great deal of their time eliciting requirements and specifications. The students had to learn this difficult lesson.

The students had many conversations with the instructor; they wrote and rewrote the requirements document several times. After finally grasping the magnitude of the requirements, the student team recognized the need for a specifications document and so had to continue the dialogue. The specifications document was written and rewritten several times as well. The instructor's approach placed the emphasis where it belonged, the requirements analysis phase of the software development life cycle. This approach also served to emphasize a key component of software development – the communication between customer and developer; if it fails so will the system [16].

FAILURE-BASED

Software engineering students must truly understand four interrelated and important concepts – risk, faults, failure, and testing. The following are adapted from Pfleeger's text [16]: Risk is an unwanted event that has negative consequences. When a software engineer makes a mistake, the human error results in a fault in the software. A failure is the departure of a software system from its required behavior. Testing must be viewed as a discovery process and the development of the test for each requirement begins during the requirements analysis phase. Risk is naturally embedded in the software development life cycle. Software engineers make mistakes that create software faults, which can lead to failure. In an academic software engineering project some of the enablers of risk are the talent of the students on the team, students dropping the course in mid-project, and the time available. Testing is critical in the discovery of faults and can help reduce risk especially if testing is continuous throughout the entire software development life cycle. In-progress reviews and rapid prototyping can be used to test understanding of requirements and specifications as well as the designs.

It has been the author's experience that real world software engineering projects are fraught with risks and faults, which can lead to failures. Testing can discover software system failure before delivery; after delivery, failure might be fatal and final. So, why should educators want academic software engineering projects that will insure student success? Many published papers

have described "successful" projects that have been implemented in software engineering courses, but not with all the required functionality or necessary testing. By real world standards, such projects are failures. So why not accept failure as a mechanism for true learning? In other words, the project in the software engineering course might not get fully implemented, but the students recognize the failure and learn from it.

Roger Schank[17] coined the phrase "expectation failure" to describe what happens when a human expects something to occur yet it fails to occur. For example, when a software development team expects a program module (e.g., reusability in the Araine-5 case) to simply be integrated into the system being developed and to work properly with other program components but it does not, the team has experienced expectation failure. Schank and others believe that expectation failure is necessary in order for learning to take place. In fact, they believe that people remember best what they feel the most. Thus when people experience expectation failure, their minds create "a reminder and a remedy." In other words, in the future they remember the circumstances surrounding this failure and avoid this type of failure by following their remedy procedure. The key idea here is that students must feel the need to internalize a reminder and a remedy. Educators can tell and show students remedies that will prevent failures; students can memorize them for a test in a course; but until a student feels the pain of failure, they will not internalize the reminder and remedy. An old adage puts it as: Seeing is believing, but feeling is real. Learning software engineering must be real. This author defines a failure-based software engineering project as one that facilitates expectation failure.

The Spiral Model[4] (see Figure 2 in Additional Readings and Notes section) is most appropriate for failure-based software engineering. First, it acknowledges the iterative nature of software development. Students felt this initially when they failed to get the requirements and specifications documents to an acceptable level for the project to progress, thus having to rewrite both documents several times. Second, the model highlights risks, constraints, alternatives, and prototyping. Starting with the initial problem scenario and sketch, students become engaged in the project realizing the unreliability of their teammates to persist in the course, their own limitations (i.e., knowledge of the problem domain, how much time they are willing to invest in the course, etc.), and the need for various courses of action that can solve the problem. A vocalized student concern was "What happens if half the team drops the course?" The instructor responded, "What can you accomplish with fewer people?" Again, educators can talk about the need to divide a complex problem into manageable sub-problems, but students must feel the need to do so. Having people drop the course or anticipating that people will drop the course creates the need to focus on the critical components of the software. The instructor's emphasis on prototyping various aspects of the development was very useful in getting students to divide the problem up and develop separate courses of action.

The instructor assigned rapid prototyping programs for the students to do individually. For example, the first prototype program was to accept a lat & long and verify that they were valid. In the real world, sensor data is usually noisy, which results in bad data composition. The noise problem with a sensor can be detected and a request for retransmission initiated. Most of the students in the instructor's class did not know what constituted a valid lat & long. This rapid prototype gave the instructor an excellent opportunity to reify a couple of software engineering activities. First, software engineers often go to the library to learn about their assigned problem domain. Second, talking about data being normal, interdependent or creating an exception is one thing, but knowing what that means is quite another and this knowledge comes from testing the prototype. All the student prototypes validated normal lat & long data. Some of the student prototypes got

all the exceptions. But none of the student prototypes got the interdependency; the input of 90 degrees, 1 minute, 0 seconds, South for a latitude was not identified as invalid. The South Pole is at latitude 90 degrees, 0 minute, 0 seconds, South; this latitude cannot be exceeded not even by one second let alone one minute. Besides being able to create moments of individual expectation failure, the rapid prototyping assignments gave the instructor an instrument that he used to evaluate the individuals on the team. A moment of team expectation failure came at an in-progress review when the lat & long validation module, which had been corrected, was coupled with the module that read the radar altitude data and the system failed.

In order to insure expectation failure, which is the basis for failure-based software engineering, a software engineering project must be in a sufficiently complex problem domain. It is also helpful if the problem domain is outside the student's normal scope of personal knowledge. Finally, the instructor must provide numerous individual and team opportunities for failure to occur.

GUIDING AND EVALUATING

In order to guide the students appropriately, an instructor must have a reasonable idea of the abilities of each. This could be difficult at a large institution, but it did not pose a problem at a small one like the author's. By the time a student is in the second year at the university, the faculty member has a pretty reliable feel as to the individual's strengths in mathematics, programming, and more. Using this knowledge, the instructor must be able to adjust the difficulty level of the project.

From the start, the instructor stressed the importance of a requirements document and pushed for its development. The instructor wanted to force the student team to deal with what they did not know. Eventually a student asked, "Given the lat & long of the launch, how do you determine the country that launched the missile?" The instructor gave a little more information: "There is a country repository that describes the border of each country via a polygon constructed by connecting sequentially stored lat & long coordinates with straight lines." If the instructor wanted to start with a simpler construct, the polygon for all the countries can be a rectangle, with the top left hand vertex being the first lat & long and the bottom right hand vertex being the second lat & long. Determining the country that launched the missile was another rapid prototyping opportunity. The instructor underscored the importance of the elicitation process, pointing out that the existence of the country repository was not in the problem scenario.

The trajectory model for the missile is also scalable, depending on the team's mathematics knowledge. If the team has weak mathematics skills, then the classical projectile model[6] (p. 343 ff) can be used. However, if the team has good mathematics skills, then a flat earth rocket trajectory[3] (p. 231 ff) would be more appropriate, and if the skills are exceptional then the solution for a round, rotating earth[3] (p. 234 ff) is the more realistic model. A rapid prototype assignment demonstrated the student's ability to use one of these models.

Regardless of the trajectory model used, the team needed to know an initial velocity for the missile. This caused a student to inquire about the different types of missiles each country might have. The instructor's response to this inquiry was to reveal the existence of a missile repository that contained initial velocities, ranges, payloads, and the countries that had each type of missile. Furthermore, for each missile and country there was another data repository containing the countries at risk. For example, Iran has the Scud B (Shahab-1) a short range ballistic missile and the countries at risk from that missile being launched from Iran are: Azerbaijan, Turkey, Pakistan, Georgia, Iraq, Kuwait, and Saudi Arabia. The instructor created these repositories based on published unclassified information [19]. Again, rapid prototyping was used to demonstrate student understanding and use of these data repositories.

The instructor's evaluation of the students, individually and collectively, was ongoing throughout the project. However, another important software engineering concept (found in the Capability Maturity Model; see Figure 3 in Additional Readings and Notes section) that students needed to experience is the evaluation of their team members. Early in the project, students were provided with an evaluation form (Table 1). The form attempted to get an "insider's view" of the team mechanics. The form was used in conjunction with individual "exit interviews" that the instructor conducted after the team presented the project deliverables in class. The instructor used the peer evaluation forms to ask very specific questions. This was a very useful tool in determining the grade for the project that each student would get. Based upon team consensus, it was obvious who the leader was and it was also obvious that the other students contributed to the best of their abilities. Perhaps knowing that this type of evaluation was going to take place, prompted students to contribute to the overall project instead of just being carried by the stronger members of the team. The last entry on the evaluation form was particularly poignant to the students.

CONCLUSION

Originally, there were six students in the software engineering course, and they worked together as one software development team. One student dropped the course shortly after the problem scenario was articulated. The remaining students "suffered" through to the end, learned a great deal, and were not at all surprised that the software could not achieve all the requirements. However, they knew which requirements had been met. Furthermore, the requirements document led nicely into the specification document, which rolled into the various levels of design with each requirement actually being traceable throughout.

The experiences of only five students do little to prove any point; however, the author believes that the project (i.e., practice) actually improved student knowledge of theory. The final grades were 1 A, 2 Bs, and 2 Cs. But the more interesting aftereffect was that the B students wanted to do undergraduate research in the missile defense domain. In sum, the author believes that failure-based software engineering projects such as the one presented in this paper demonstrate a valuable approach to teaching software engineering and prepare computer science graduates to deal with software development when things do not go according to plan due to risk, faults, and failure.

REFERENCES

1. Ben-Ari, M. (1999) The Bug That Destroyed a Rocket. *Journal of Computer Science Education*, 13, 2, 15-16.

2. Ben-Ari, M. (2001) The Bug That Destroyed a Rocket. *SIGCSE Bulletin - inroads*, 33, 2, 58-59.

3. Bennett, W. (1976) *Scientific and Engineering Problem-solving with the Computer*. Prentice-Hall, Inc.: Englewood Cliffs, NJ.

4. Boehm, B. (1988) A Spiral Model for Software Development and Enhancement. *IEEE Computer*, 21, 5, 61-72.

5. Bracken, B. (2003) Progressing from Student to Professional: The Importance and Challenges of Teaching Software Engineering. *Journal of Computing Sciences in Colleges*, 19, 2, 358-368.

6. Cruse, A. and Granberg, M. (1971) *Lectures on Freshman Calculus.* Addison Wesley Publishing, Inc.: Reading, MA.

7. Dooley, J. (2003) Software Engineering in the Liberal Arts: Combining Theory and Practice. *ACM SIGCSE Bulletin – inroads*, 35, 2, 48-51.

8. Gertz, B. (2003) Failed Missile-defense Test Probed. *The Washington Times* (June 20) http://www.washingtontimes.com.

9. Graham, B. (2001) *Hit to Kill.* Public Affairs: New York, NY.

Evaluation submitted by: _____ Date:

Evaluation of: _____

On the back of this sheet of paper, comment freely on any and all matters regarding the evaluation of the above named individual.

Using the Requirements Specification document as a point of reference, list the specific test data that the individual developed in whole or in part for what required functionality.

Using the Technical Design document as a point of reference, list the specific components that the individual designed in whole or in part.

Using the Program Design document as a point of reference, list the specific modules that the individual programmed in whole or in part.

The scoring scale is:	Unacceptable	Poor	Fair	Good	Very Good
	1	2	3	4	5

Evaluate the individual's:

knowledge of the application _____

knowledge of the C++ programming language _____

knowledge of the tools being used (e.g., UNIX, X-windows, etc.) _____

ability to communicate with others (e.g., has an attitude, listens, gives clear instructions) _____

ability to share responsibility with others (e.g., blames others, has integrity, exhibits fairness) _____

work ethic (e.g., not reliable, has to be told to do everything, must be supervised, self-starter) _____

NOT including yourself, rank order ALL the members of your team in overall performance and contributions to this project (1 is the best, 2 is the second best, etc.). NO TIES ALLOWED.

Based upon the individual's performance on this project, select ONE:

_____ Promote _____ Keep on team _____ Terminate

Table 1. Form for Peer Evaluation

10. Hamlet, D. and Maybee, J. (2001) *The Engineering of Software: Technical Foundations for the Individual*. Addison Wesley Longman, Inc.: Boston, MA.

11. Isbell, D., Hardin, M., and Underwood, J. (1999) Mars Climate Orbiter Team Finds Likely Cause of Loss. NASA Headquarters, Washington, DC Release 99-113 (September 30) http://mars.jp1.nasa.gov/msp98/news/mco990930..

12. Isbell, D. and Savage, D. (1999) Mars Climate Orbiter Failure Board Releases Report, Numerous NASA Actions Underway in Response. NASA Headquarters, Washington, DC Release 99-134 (November 10, 1999) http://mars.jpl.nasa.gov/msp98/news/mco991110.html.

13. Jones, C. (2002) Defense Software Development in Evolution. *CrossTalk: The Journal of Defense Software Engineering* (November) http://www.stsc.hill.af.mil/crosstalk/2002/11/jones.html.

14. Lopez, A. (2003) Increasing African American Participation in Department of Defense Research. *Proceedings of ADMI 2003 Conference*, Washington, DC, 16-23.

15. Polack-Wahl, J. (2003) Software Engineering: A New Approach for Small Departments. *Journal of Computing Sciences in Colleges*, 18, 3, 26-31.

16. Pfleeger, S. (2001) *Software Engineering: Theory and Practice* (2nd Edition). Prentice Hall, Inc.: Upper Saddle River, NJ.

17. Schank, R. (1997) *Virtual Learning*. McGraw-Hill: New York, NY.

18. Sommerville, I. (2001) *Software Engineering* (6th Edition). Pearson Education Limited: Harlow, England.

19. Spencer, J. (2000) *Ballistic Missile Threat Handbook*. The Heritage Foundation: Washington, DC.

20. Stiller, E. and LeBlanc, C. (2002) Effective Software Engineering Pedagogy. *Journal of Computing Sciences in Colleges*, 17, 6, 124-134.

21. Villarreal, E. and Butler, D. (1998) Giving Computer Science Students a Real-World Experience. *ACM SIGCSE Bulletin – inroads*, 30, 1, 40-44.

ADDITIONAL READINGS AND NOTES

Andriole, S. (1993) *Rapid Application Prototyping: The Storyboard Approach to User Requirements Analysis.* John Wiley: New York, NY.

Davis, A. (1995) *201 Principles of Software Development.* McGraw-Hill: New York, NY.

Pressman, R. (1997) *Software Engineering: A Practioner's Approach* (4th Edition). McGraw-Hill: New York, NY.

Saiedan, H. and Kuzara, R. (1995) SEI Capability Maturity Model's Impact on Contractors. *IEEE Computer*, 28, 1, 16-26.

Thomas, B. and Duggins, S. (2002) The Internationalization of Software Engineering Education. *Proceedings of the 2002 American Society for Engineering Education Annual Conference & Exposition*, Session 2260.

Wasserman, A. (1996) Toward a Discipline of Software Engineering. *IEEE Software*, 13, 6, 23-31.

BIOGRAPHICAL INFORMATION

Antonio M. Lopez, Jr. has held the Conrad N. Hilton Endowed Chair in Computer Science at Xavier University of Louisiana since July 1, 2000. During the 2000-2001 academic year, he also held the Chair in Artificial Intelligence at the United States Army War College, Carlisle, PA in the Center for Strategic Leadership. He

was recognized for his work there with the Department of the Army Superior Civilian Service Award, and he continued in dual status as Visiting Professor in Artificial Intelligence until July 2003. At Xavier he has taught a variety of upper-level undergraduate computer engineering and computer science courses including software engineering. Lopez received his ph.D. in Mathematical Sciences from Clemson University in 1976. His current research areas are: intelligent agents, knowledge-based systems, ontology development, and knowledge management. Email: tlopez@xula.edu. WebPages: www.xula.edu/~tlopez.

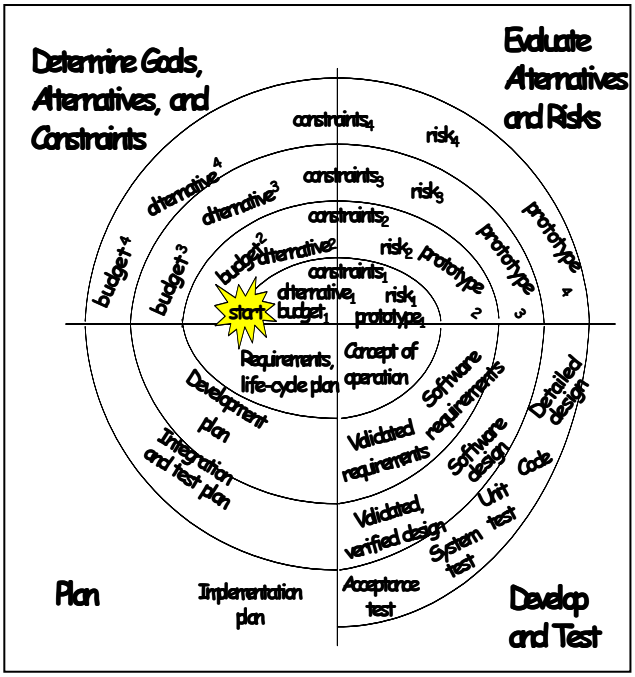

Figure 2. The Spiral Model

The graphic presented in Figure 2 above is adapted from Pfleeger's text (2001). The Spiral Model shows that the process of software engineering is iterative. The instructor used four spiral bands because students had about four months to develop their software product. The model also underscores two very important concepts -- risk and rapid prototyping. Finally, the model depicts change, with new constraints and alternatives being introduced in each spiral band. The students in the course were not required to develop or adhere to a budget, but the model clearly shows that a real world application must have a budget review in each spiral band.

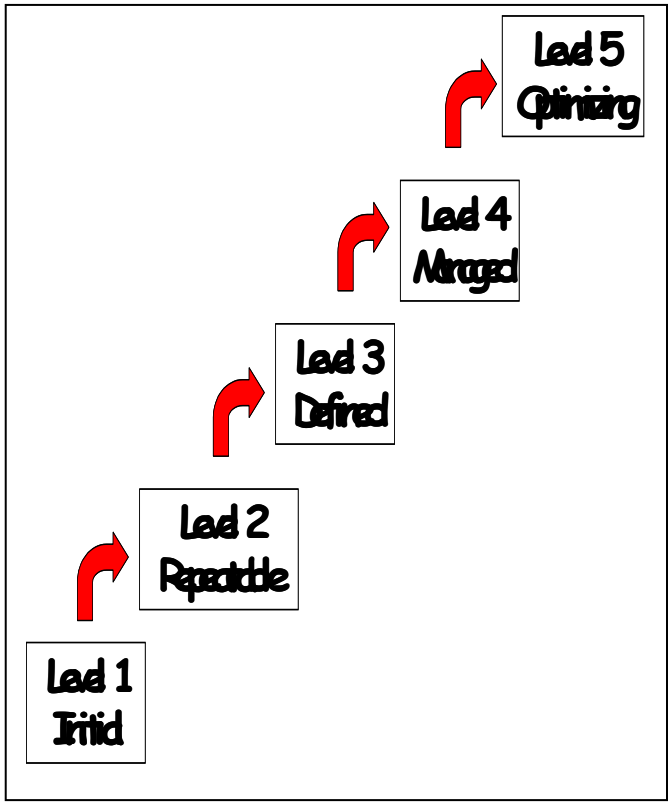

Figure 3. The Capability Maturity Model.

The graphic presented in Figure 3 above is adapted from Hamlet and Maybee's text (2001). The Software Engineering Institute (SEI) at Carnegie Mellon University developed the Capability Maturity Model (CMM) to assist the Department of Defense in assessing the quality of its contractors. Evaluation and reflection are key components in moving from one CMM level to another. Corporations at CMM level 5 have optimized the software engineering process whereas corporations at the CMM level 1 have not yet evaluated their procedures nor reflected upon how to go about improving them to the point that the successes are repeatable.

For more information on SEI and CMM see: www.sei.cmu.edu/cmm/cmms/cmms.html.