

A graduate degree in software engineering, and why you should consider it

Lewis Berman, Ph.D.
Loyola University Maryland

October 14, 2013

Software development and maintenance projects, large and small, have been performed since at least the 1960's. Still today, too much software is completed behind schedule and of lower than desired quality. Some software efforts are abandoned completely. Why? A lack of effective software engineering. Part of the problem is that Computer Science and Information Technology programs do not, in general, prepare professionals for the challenges inherent in being a software engineering leader.

This white paper explores a range of software engineering issues and suggests that you can help solve them by expanding your software engineering knowledge and skill. You will learn why a graduate program specific to software engineering is the answer of choice and how Loyola's Software Engineering graduate program fulfills that promise. You should find the paper relevant whether you are a software practitioner or a corporate manager of software development projects.

1 The State of Software Engineering

Building a high-quality software product or system is a difficult endeavor. Many choices must be made. Who will be on the project team? How much will the project cost? Should the project perhaps be built incrementally using agile methods, or is a different approach better? What will the system's architecture look like? What should we do about high-risk areas such as integration with new hardware? What language should we choose and why? How do we verify that the program meets the users' expectations and needs?

Bett Correa, a software architect at a major telecommunications company, tells us in an article in IEEE Software [10] that she spent the early days of her career using the approach, "code as fact as possible and hit compile to see if you get lucky." She adds that getting lucky was not a common occurrence. She indicates that, as she was progressing into her role as an architect, something that did not work was "doing architecture without training." So she obtained training and experience. Ms. Correa describes her recent work as using a true engineering approach: "I had to decide several parts of the design, such as which system should do which function when two systems were equally equipped to handle that function. Many times, I found that there wasn't a single right answer: my design decisions were based on trade-offs, such as giving something more security but allowing it to take longer to do a specific function. One decision might work well in most cases but fail in others, so I had to figure out in how many cases it would fail and whether the workaround for those few cases would end up being expensive."

Too many organizations to this day employ the no-brainer "code and test" paradigm. While that may work (sometimes) for very small programs, even then such an approach does not ensure that the user's expectations are met. Forty or so years ago, an improvement to "code and test" appeared in the form of the "waterfall model": figure out the requirements, do the design, code the project, then test it and release it. That model never worked cleanly in practice, so others such as Boehm's Spiral Model [7] and the Software Productivity Consortium's Evolutionary Spiral Model [11] were introduced. Agile [13] has been increasingly popular in the 2000's. Since Agile is still something of a work in progress, practitioners need to keep up with its evolution. Additionally, there are a number of variants of Agile; for example, Scrum versus other team-oriented methods. Contrary to popular beliefs, in many situations Agile is not the most cost-effective or highest-quality choice. A project manager or chief engineer needs to possess a strong general working knowledge of software engineering principles and supplement that knowledge with particular state-of-art practices.

Consider Figure 1, an adaptation of a graph introduced by Barry Boehm circa 1980 and still quite relevant today. It shows that the cost of finding an error late in a software development project is exponentially higher than finding it earlier. (Note that the Y axis is logarithmic.) For example, suppose your team gets a requirement wrong right up front, and nobody detects that until the product or system is delivered. That likely translates into a major rewrite of a significant portion of code and a delay in the project schedule.

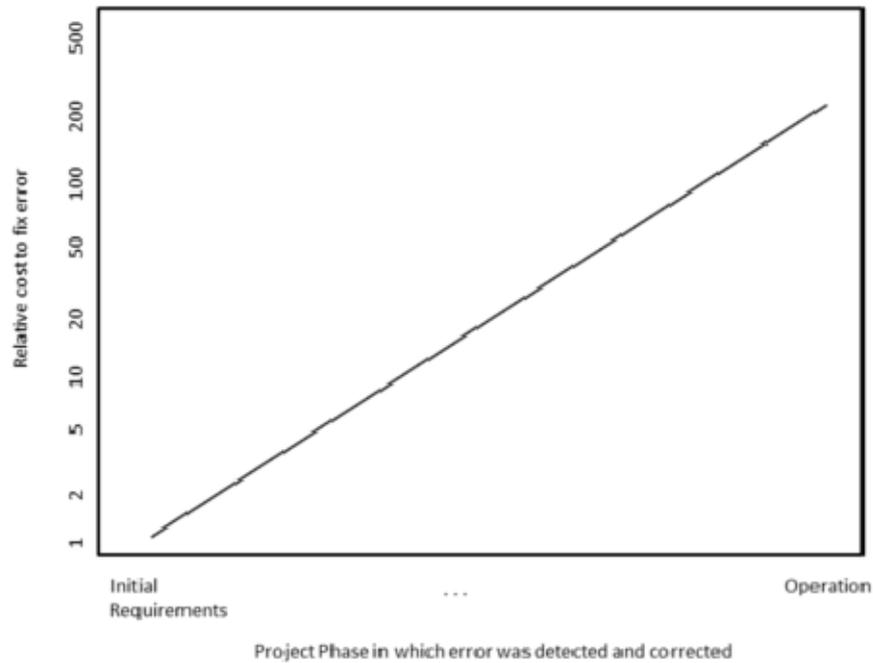


Figure 1: Cost of Error Discovery in Project Timeline

The cost could be 200 times or more higher than having detected the error early on, before anything was designed or coded. The added cost would have been avoided had knowledgeable project leaders put the proper controls in place throughout the development cycle. The organization could have helped ensure that by having reasonable standards and practices in place.

No development methodology immune to such a cost overrun. In fact, agile methods introduce their own variants of such a risk. A big advantage of agile methods is effectively developing a system when it is unclear what that system should do. Suppose it is *clear* what a system should do prior to its development. And, suppose the features of that system are implemented piecemeal in a series of agile iterations. Down the road, a feature is encountered that requires redesign of the system's architecture. That may end up costing more than having done a considered design up front, taking all of the features into consideration. Or possibly, simply better planning the contents of each Scrum may also have alleviated the situation. Was a set of planning guidelines in place? Did the organization require the project engineer to

consider project lifecycle alternatives?

Thus, software engineering is about architecting robust systems, adopting intelligent software practices, and using those processes correctly. It is also increasingly about empirically validating such practices and continually improving them.

2 Software Engineering Content at the Graduate Level

Loyola University Maryland offers a Software Engineering graduate program, culminating in the degree *Master's of Software Engineering (MSSE)*. It is one of relatively few such programs in the U.S., having been inspired by the first such program at Carnegie Mellon University. While the Software Engineering program differs from Loyola's more traditional *Master's of Computer Science (MSCS)* program, the two have content and courses in common. The Software Engineering program assumes that the student has been previously versed in programming and related computing topics, typically by holding a bachelor's degree in Computer Science or a related field. While the Software Engineering program is targeted to professional software developers, it can also be taken by students continuing directly from a prior degree program.

Traditional Computer Science programs teach technical aspects of building software: data structures, algorithms, design patterns, and the like. They may touch on software engineering practices, but they generally do not explore those in depth. Loyola's graduate program in Software Engineering covers such topics in greater detail with a combination of instruction and hands-on practice. Some of the areas covered include

- *Requirements Specification and Management* - Many issues revolve around requirements. What is a good requirement? How do we know that a set of requirements is complete? How do we maintain an organized and relatively stable set of requirements? What are the ramifications when a requirement has to change? How can we most effectively employ tools such as the Unified Modeling Language (UML) [8] in support of requirements analysis?
- *Configuration Management and Control* - Most organizations have discovered that software under team development has to be kept under

version control using a tool such as Subversion [5] or Git [3]. Even so, questions remain. Is the project using the correct collection of baselines? Are their repositories organized efficiently? Who oversees whether the repositories are correct? Is there a disaster recovery plan?

- *Software Verification and Testing* - test planning and sequencing; verification of untestable but otherwise verifiable features.
- *Software Architecture* - the organization and layering of large-scale software. How the architecture can promote or inhibit effective team development; how the architecture supports the -ilities (see below).
- *Software Systems Engineering* - integration of the software product with hardware, sensors, and other systems; design of the software product to make such integration smooth. For example, path-oriented integration declares that the product will be integrated along the lines of end-to-end usage.
- *Software Metrics* - how to measure quality of product while it is still in progress; what to measure and what not to measure in order to be unobtrusive; how to know the project is on track in terms of cost and schedule. For some examples, see [1].
- *Software Maintenance Practices*. Software maintenance may begin even prior to the first formal release of a software product. How does maintenance differ from initial development of a product? What documentation is necessary? How much documentation is too much?

2.1 Software Engineering Management

Management techniques and issues certainly apply to leadership in software engineering. A software project manager may lead up to 50 or more people, a team lead may lead up to a dozen or more, and a technical director or project engineer will interface with all the people on the project and many outside of it. Even a small project or company will have a need for effective management. Tools such as Gantt and Pert charts are likely to be utilized. Many organizations will want to have personnel certified as Project Management Professional (PMP) by the Project Management Institute [4].

However, standard management practices cover only part of what a leader in software engineering must know and do. For example, many organizations

implement best practices according to the Software CMMI [9], with which the leader should be familiar. As a related example, the CMMI spells out the need for software project cost and schedule estimation. There are several major ways to implement this, including software-specific, top-down, parametric estimating using industry tools. These examples are briefly explored below.

2.1.1 Software CMMI

Many commercial companies, governmental organizations, and government contractors seek to conform to advanced software process levels of the asdf asdf asdf (CMMI). Government contractors are often required to conform to the CMMI, while other organizations will consider adopting it as a matter of good practice. In addition to good practices, the CMMI calls for continuing organizational software process improvement. Both the CMMI itself and software process improvement are addressed in Loyola's Software Engineering graduate program.

2.1.2 Software Project Estimation

Many projects in industry as a whole are estimated in a bottom-up manner, a practice having had its origins in constructing buildings. You count the number of bricks and know about how many a bricklayer can lay per hour; you know about what it will take to put in each plumbing fixture, etc. Software projects can be estimated in this manner, but software (i.e., information) does not neatly fall into "number of bricks." The software industry has developed top-down, parametric methods of estimation, based on agreed-upon measures such as "equivalent source lines of code (ESLOC)" and "function points." Such methods can account for the desired level of risk to be taken. Top-down methods can be used exclusively or in conjunction with corroborating bottom-up methods.

2.2 -ilities

The "-ilities" are the hallmarks of a well-designed versus poorly-designed software system. *Usability* is the ease with which a user can interact with the system. An entire course, CS 774 Human-Computer Interaction, revolves around usability. *Maintainability* is the ease with which the system can be

corrected or modified once it is operational. Maintainability ties into many aspects of software engineering such as the system's architecture, level of requirements maintenance, level of documentation, and consistency of coding. *Scalability* is how well the system can be scaled to larger environments, more servers, more users, etc. And so forth.

The "-ilities" exist at all levels and orthogonally across the system or program. For example, we want to make a transaction, data, or the entire system state *recoverable*. We think of a single, good requirement as *testable* or at least *verifiable*.

2.3 Security

Software security is an essential part of any user-based product or system. It differs from network security and enterprise security in that security features are built into the code itself. A malicious hacker may be able to break into the network and gain control of a server, but to render a program's operation incorrect, the hacker must be able to take advantage of some exploit such as buffer overflow. Coding securely and using other good software engineering practices in general can radically reduce the number of such exploits. It cannot be emphasized enough that *good software security is the result of well-engineered code*. Courses dealing with network and enterprise security are available in the Software Engineering program as electives.

3 Loyola's Software Engineering Program

The Master's of Software Engineering is an evening program, taught at Loyola's two graduate centers for the convenience of working professionals. Each class meets one evening per week. The program is taught by a combination of full-time members of the Computer Science Department and affiliate faculty who are experts in their disciplines. For example, Dr. David Binkley is a full-time professor and researcher in automated program documentation through text extraction. Conversely, Michael Berman is a project engineer at Northrop Grumman who leads a project building advanced, in-helmet displays for fighter pilots. He is also involved in corporate software process improvement initiatives. Our faculty are also integrally connected to the software engineering research community. Dr. Raunak of the full-time faculty has recently served as co-chair of the Empirical Software Engineering

and Management conference [2].

CS 770, Software Engineering, is the pivotal course on which the advanced Software Engineering courses are based. Advanced courses include Engineering Systems Analysis, Object-Oriented Analysis and Design, Software System Specification, Human-Computer Interaction, Software Reliability and Testing, Software Architecture and Integration, Software Cost Estimation and Management, and Software Maintenance and Evolution. CS 762, Database Systems, provides practical experience with relational databases and the relational theory that goes along with that experience. In addition, electives can be chosen from advanced courses in the Computer Science program, including Special Topics courses which cover a range of pertinent technologies.

3.1 Advanced Topics

Warning! The Software Engineering graduate program may expose you or your employees to ideas you haven't previously heard of. You could become a pioneer bringing to your organization program slicing [12] or software sonification [6]. How about empirical software engineering - choosing practices based on empirical studies. Or *mutation testing* [?], or even *exhaustive testing* [?]. Then there's model based programming, information retrieval methods for automatic program documentation, . . .

These topics are often presented in Special Topics courses. If you're interested in a topic that isn't covered in a course, or you want to go into more depth with one that is, you can take an Independent Study with one of our full-time or affiliate faculty. Independent Study results have sometimes been published in software-related journals and conference proceedings.

4 Summary

There is a great need for software designers, team leads, project engineers, and managers who possess deep understanding of software engineering. Loyola's answer is its Software Engineering graduate program, culminating in a Master's of Science in Software Engineering (MSSE). This degree can help distinguish you or members of your technical staff as among the elite.

Interested? Check out the Computer Science Department's web site at <http://www.loyola.edu/cs>. Click on the *graduate* link. From the *Graduate Portal Page*, you can get to course listings and descriptions, application

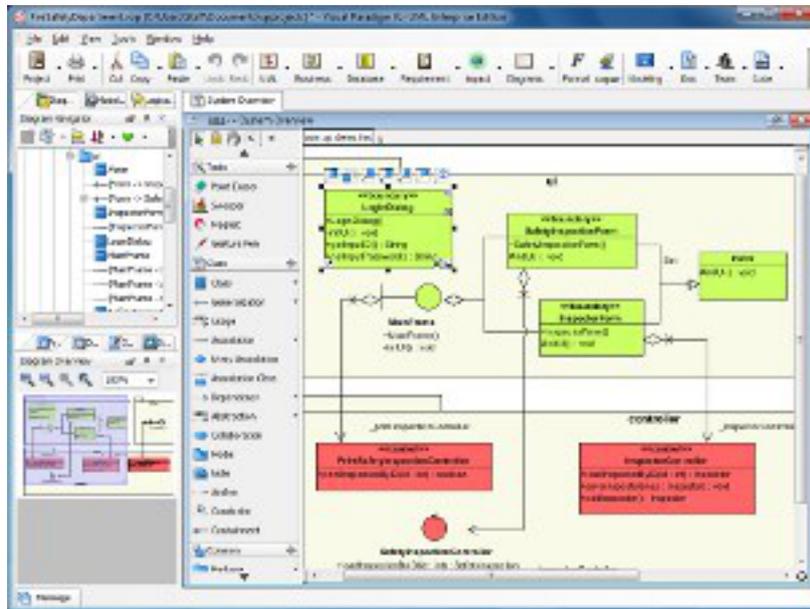


Figure 2: UML software design tool displaying a class diagram

requirements, an online application form, and more.

We would love for you to engage us in further discussion. Are you intrigued by our program? Do you have suggestions for content we haven't yet incorporated? Give us a shout at inquiry@cs.loyola.edu or (410) 617-2587.

References

- [1] CSUN software metrics guide. Last accessed 5 October 2013. [Online]. Available: <http://www.ecs.csun.edu/rlingard/comp589/ProjectMetrics.htm>
- [2] Empirical software engineering and management. [Online]. Available: <http://esem-conferences.org/>
- [3] Git. Last accessed 5 October 2013. [Online]. Available: <http://git-scm.com/>
- [4] Project management institute. Last accessed 5 October 2013. [Online]. Available: <http://www.pmi.org/>

- [5] Subversion. Last accessed 5 October 2013. [Online]. Available: <http://subversion.apache.org/>
- [6] L. Berman and K. Gallagher, “Using sound to understand software architecture,” in *Proceedings of the 27th ACM International Conference on Design of Communication (SIGDOC’09)*, Bloomington, USA, October 5-7 2009, pp. 127–134.
- [7] B. W. Boehm, *Software engineering economics. 1981*. Prentice-Hall.
- [8] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Reading, USA: Addison Wesley, October 20 1998.
- [9] CMMI Product Team, “CMMI for development, version 1.3 (CMU/SEI-2010-TR-033),” Software Engineering Institute, Pittsburgh, USA, Tech. Rep., November 2010.
- [10] B. Correa, “How are architects made?” *Software, IEEE*, vol. 30, no. 5, pp. 11–13, 2013.
- [11] R. Cruickshank, J. Gaffney, and R. Werling, “Process engineering with the evolutionary spiral process model,” Software Productivity Consortium, Herndon, USA, Tech. Rep., December 1992.
- [12] K. B. Gallagher and J. R. Lyle, “Using program slicing in software maintenance,” *Software Engineering, IEEE Transactions on*, vol. 17, no. 8, pp. 751–761, 1991.
- [13] G. Melnik, P. Kruchten, and M. Poppendieck, in *Agile2008 Conference*, Toronto, Canada, August 4-8 2008.