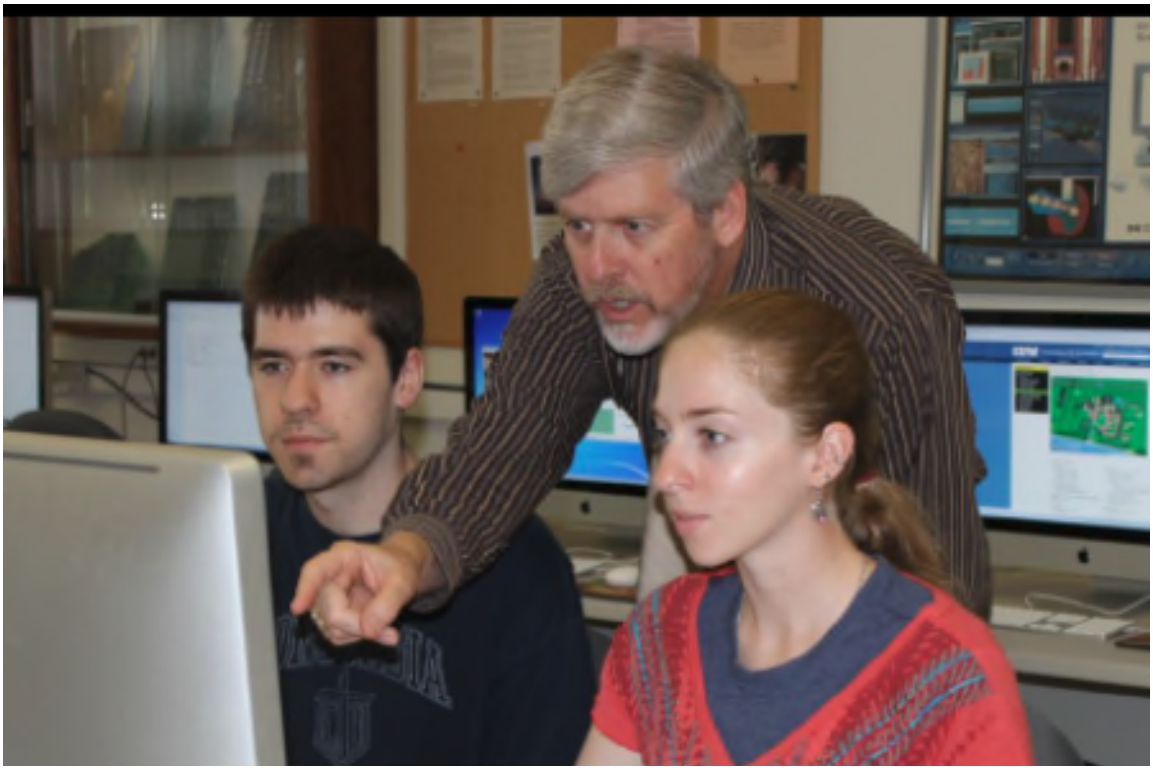
Software Engineering: The Idea, Implementation, and Interaction of Professional Software Creation

Prof. Gary Locklair, PhD

Software Engineering isn't a *part* of Computer Science. Software Engineering **is** Computer Science. In a very real sense, the heart and core of Computer Science is found in Software Engineering. Computer Scientists solve problems. Software Engineers employ professional methods to solve complex problems well. Computer Science concerns creating automated solutions to real world problems using computer systems. The systems we create consist of hardware, software, and people. So, why focus on software? This text will help you answer that question ☺ Why am I capitalizing Software Engineering and Computer Science? I suppose in order to make a point. We'll see how well I do that in this text.



CSC 370 students in action 1

Part I – Idea

0 – Introduction to Software Engineering (part A)

What's the difference between Oliver Gavin, race car driver, and Gary Locklair, race car driver? Oliver Gavin is a member of Team Corvette Racing. He currently drives the number 4 C7.R Corvette race car. He is a four time ALMS champion with 40 class victories. He also has five class wins in the 24 Hours of Le Mans, including the 2016 race. Gary Locklair owns a 1990 Corvette ZR-1 that he has driven on several road courses.

The difference is Oliver Gavin is a professional and Gary Locklair is an amateur. (It should also be noted that Gavin's C8.R produces 500 horsepower from 5.5 liters of engine while Dr. Locklair's ZR-1 manages 460 horsepower from 5.7 liters.)

Corvette Racing's C8.R Corvette #3 unveiled in 2020

What's the difference between a professional and an amateur? In this scenario you could cite many items. As one example, Gavin has a state-of-the-art race car. As another, there's a large team of people supporting him. Thirdly, he has thousands of hours of practice. Finally, he is also a very good driver.

What's the difference between CUW's CSC 200/250/300 courses and CSC 370? In a nutshell, it's this: CSC 200/250/300 deal with coding. Coding is programming-in-the-small. CSC 370 deals with software engineering. Software engineering is programming-in-the-large.

It's one thing to take your car to Road America on a Saturday morning to participate in a HPDE (High Performance Driving Event). That's the kind of thing amateurs like me do all the time. It's quite another thing to bring the Corvette C8.R race team to Road America for the American Le Mans series. That's the kind of thing professionals do.

Road America race course http://www.road 1

From one perspective, there are many things that are the same in the two racing scenarios. The 4 mile race course at Road America is the same. Both of the drivers are in Corvettes. I'm sure we could find a hundred similarities between them. How much difference could there really be? The answer is, an enormous amount.

How much difference is there between coding and software engineering? The answer is, an enormous amount. Software engineering isn't just about coding. Software engineering is about learning how to think as a computer scientist. Software engineering is about developing systems in a professional manner. Coding, like amateur racing, is done for fun. Software engineering, like professional racing, is done to win (I avoided FTW).

Developing software in the context of software engineering is different from coding. Development includes *all* aspects of the project from conception to retirement. In order to develop software, the engineer must understand the problem, produce requirements, develop specifications, create the design, and validate and verify the project, among a host of additional tasks. The coding or programming is one aspect of a much bigger and more interesting picture.

It's your second week on the job at LG industries. Your training is over and your supervisor calls you into her office. "We've secured a $350 million contract to transition Amazon.com's EFT transaction software to a Cloud 3.0 platform. You'll work with the architecture team to design the framework." How in the world would you solve this huge, complex problem? Where would you begin? Should you hack out some code in C# or Python to show your supervisor?

What is Software Engineering? Software Engineering deals with software development. The word development implies more than just creating some code! Computer science is problem solving. Coding is problem solving. Software Engineering is *professional* problem solving. Professionalism doesn't just mean getting paid to do the job. Professionalism means doing the job right and well. It

means being consistent, efficient, productive, and effective. Software Engineering provides a framework for doing the software development job well.

I believe that there are three ways people view work. Some people have a job. A job is just a means to generate income. There's nothing wrong with making money, of course, but a job is likely not a long term commitment. I spent some time working as a mechanic in a Ford garage during college. It was a way to help pay tuition. I never planned on being a mechanic my whole life - now I just do it for "fun" ☺. Other people have a career. A career is both a means to generate income and derive personal satisfaction. I spent nearly 10 years working in the computing industry. Much of my career was with Hewlett-Packard company where I held various positions, including software engineer. Finally, all Christians have a vocation. A vocation is a calling from God. A vocation is a means to love and serve other people, and not just a way to line your pockets with money. Martin Luther said that Christians are meant to be "God's masks" in the world. In other words, our vocation is to serve other people no matter what our particular profession is. When we serve others, they see the hand of God's providence through our actions. The primary focus of a software engineer's calling is to solve problems well for other people. The reason I left working in industry (and the large paychecks it provided) to teach at a Lutheran university was to serve other people.

While reading about software engineering is interesting (OK, it was interesting to me when I was writing it, at least), you can't learn software engineering passively. You have to **do** software engineering to really understand the principles and concepts. It's time.

what is software engineering?

Abstractly, software engineering is professional problem solving. Concisely, software engineering is the management of the entire software development process from the conception to the retirement of a product. The goal of software engineering is to solve a problem by creating an acceptable product for the user. A successful software engineering project is one which produces an acceptable product. In order to be successful, professional software engineers follow a defined process that guides them in the development of software systems.

Ultimately the *user* will determine whether the product produced is acceptable or not. Software engineers, like all computer scientists, solve problems for *other*

people. We don't create systems for our own amusement. No matter how much we like the system, no matter how stunning our code is, no matter how richly we've documented the product, it is the end user that determines whether we have successfully created a good solution to the problem. The purpose of the software engineering process is to help ensure that we do indeed develop acceptable products.

In a simple sense, problem solving involves four aspects. First, you must understand the problem that needs to be solved. Second, you must plan how to develop the solution. Third, you must create the proposed solution. Fourth, you must test that the solution actually solves the problem. While this simple model underlies software engineering, the process is deeper and more involved (as you would expect). Unlike an amateur, a professor understands and follows a defined problem-solving process. To solve problems, a software engineer has to think deeply. Solving problems is hard work. A problem-solving framework (such as Software Engineering) makes the task more manageable.

A computer system is a combination of three components, hardware, software, and people, all working together to solve a problem. The hardware provides the capabilities of the system; in other words, it determines what is possible. For example, in order to generate a 3D image, a 3D monitor is required. If the hardware doesn't support 3D, the image will only be 2D. The software provides the abilities of the system; in other words, it determines what is plausible and likely. For example, even if we have a 3D capable display, the system will not produce 3D images unless the software turns that possibility into reality. Some people, such as software engineers, are developers. Their job is to create the problem-solving tool. Other people are users. Their job is to employ the tool in production situations in order to solve specific problems in their business.

All creative processes have three aspects: idea, implementation, and interaction. These are not linear steps, but aspects of creativity that intersect each other when an engineer (or any creative person) is developing a product. An idea is a mental concept, or awareness, of the problem and its possible solution. An idea refers to a serious examination and contemplation of the situation. In order to understand things, we develop ideas, or mental images. Implementation turns the idea into reality. While an idea can exist in the mind, a product has to be implemented before it can be considered, assessed, and used. Implementation instantiates our mental picture into something concrete. The implementation is (hopefully) a true representation of the original idea. After implementation, the idea is put into effect and its promises are fulfilled. Without interaction, both the idea and the implementation are impotent. If there is no interaction with the system, the system

is useless. The purpose of a system is to solve problems and this effect is only realized via interaction. This Trinitarian nature of creativity was originally proposed by Dorothy Sayers. I have adapted her model to apply to computer science.

The actual software engineering process used in different companies by different teams varies. There exists no single correct software engineering process. The process we will follow has 6 major phases: requirements, specification, design, instantiation, execution, and maintenance. While the terms and details vary widely, all software engineering processes include aspects of these fundamental phases.

As a whole, software engineering encompasses issues related to project, process, and product. A project is a planned undertaking. The development of an inventory control system would be an example of a project. A process is a series of activities directed to an end. The software engineering management techniques used to develop the inventory control system form a process. A product is the result of the development effort, the created software system. We consider the program itself to be a part of the product, but remember the code is only one aspect of the final, finished product.

Margaret Hamilton with the code her team developed for Apollo 11

a little deeper

Here's how dictionary.com defines software engineering, "A systematic approach to the analysis, design, implementation and maintenance of software. It often involves the use of CASE tools. There are various models of the software life-cycle, and many methodologies for the different phases."
[http://dictionary.reference.com/browse/software+engineering]

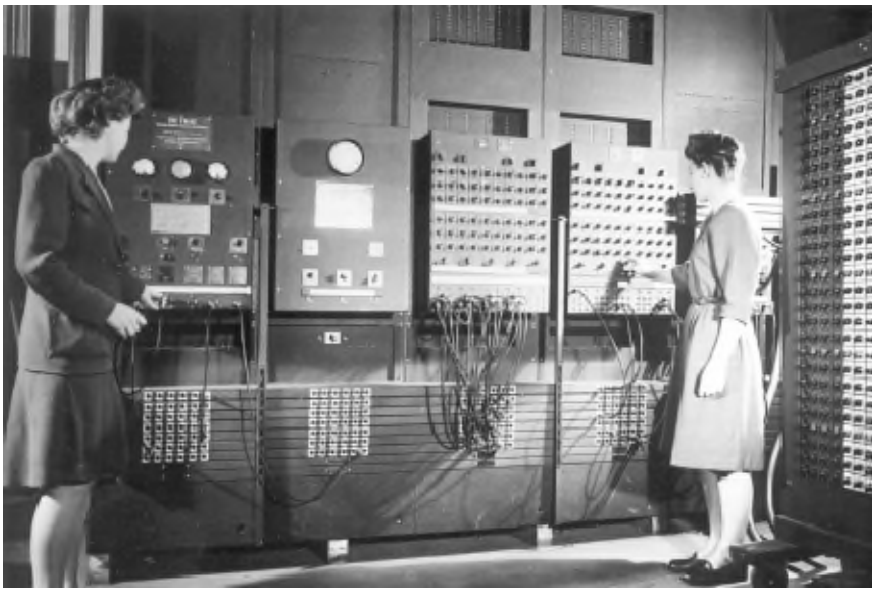
The Merriam-Webster online dictionary states software engineering is, "a branch of computer science that deals with the design, implementation, and maintenance of complex computer programs." [http://www.merriam-webster.com/dictionary/software%20engineering]

H. D. Milis of IBM's Federal System Division stated, "Software engineering may be defined as the systematic design and development of software products and the management of the software process. Software engineering has as one of its primary objectives the production of programs that meet specifications, and are demonstrably accurate, produced on time, and within budget."
[http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5387924&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F5288519%2F5387922%2F05387924.pdf%3Farnumber%3D5387924]

The ISO/IEC/IEEE Systems and Software Engineering Vocabulary (SEVOCAB) defines software engineering as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software." [SWEBOK, 2014 edition and www.computer.org/sevocab]

The Ralston and Reilly *Encyclopedia of Computer Science* states "Software engineering is the disciplined application of theories and techniques from computer science to define, develop, deliver, and maintain, on time and within budget, software products that meet customers' needs and expectations. Software products include the actual program source code and data structures, as well as the documents necessary to produce these, and documents and interface programs necessary to use them in the intended environment."
[http://dl.acm.org/ralston.cfm?CFID=459427835&CFTOKEN=58119723#]

In the beginning (the 1960s), software development was an art form. Most programmers did their own thing. If you recall your history, the first true computer, the ENIAC, was completed in 1946. The first "programmers" were a team of women from the University of Pennsylvania. In order to create instructions for the ENIAC, the programmers had to read and understand circuit diagrams in order to connect the various sections of the hardware together with wires. The ENIAC Programmers Project records,
"In 1946 six brilliant young women programmed the first all-electronic, programmable computer, the ENIAC, a project run by the U.S. Army in Philadelphia as part of a secret World War II project. They learned to program without programming languages or tools (for none existed)—only logical diagrams. By the time they were finished, ENIAC ran a ballistics trajectory—a differential calculus equation—in seconds! Yet when the ENIAC was unveiled to the press and the public in 1946, the women were never introduced; they remained invisible."
The machine these women programmed, the ENIAC, was not initially known as a computer. In the 1940s the term "computer" referred to people, not machines!



http://eniacprogrammers.org/press-info/ 1

Jean Jennings Bartik (left) and Frances Bilas Spence (right) preparing for the public unveiling of ENIAC, Feb 1946

One of the women involved, Jean Bartik, said "For many years in the computing industry, the hardware was it, the software was considered an auxiliary thing." [http://googleblog.blogspot.com/2008/12/jean-bartik-untold-story-of-remarkable.html]

In the early days of computing, software was considered a secondary aspect of the system. While hardware was engineered, software was put together in an ad-hoc fashion. There were no guidelines or "best practices" available. Today, we would find this attitude bizarre, but it was common from the beginning of modern computing until the 1960s.

By the 1960s many people realized there was a "software crisis." Software development suffered from many problems, including projects that were behind schedule (they were late), projects that were over budget (they cost too much), and products that were full of bugs (they didn't work well). Many felt that the software crisis was the result of an undisciplined software development process. It is unfortunate that 55 years later this same "unholy trinity" of software issues (behind schedule, over budget, full of bugs) still plagues software development.

There's nothing wrong with software development being an art form. There is definitely an artistic aspect of software development. However, many people incorrectly assume that "art" is merely self-expression and that "art" is something that has no rules and follows the whims of the artist. Genuine art is beautiful, good, and true. The aesthetic aspect of good art is recognizable and edifying to all. There is an art to software engineering (and there are additional important aspects also). Coders sometimes believe that no matter how they code, as long as the product

works in the end, everything is fine. I've heard coders lament, "Don't tell me how to write code. Coding is an art form. If you give me rules, you'll stifle my creativity." This isn't true. Even master artists follow "rules" in order to produce great works of art. [See Software as Art, *Communications of the ACM*, Volume 48 Issue 8, August 2005 Pages 118-124]

Programming is a process. Programming is the process programmers pursue when using a programming language to produce programs (did you count the number of "pro" words in this sentence?). Even if you don't think you are following a process, you are. Rather than follow an ad-hoc process, why not use a process that will make the job more effective? Professional programmers follow a defined process when they develop software. To an amateur, the process looks difficult, time-consuming, and foolish. Professionals know that following a process will make the development easier in the long run. Amateur race car drivers don't want to "waste time" studying a track to find its "line," they just jump in and start driving as fast as they can. A professional race car driver will physically walk around a race course before ever driving it, looking for the "line" that gets around the corners the fastest. While finding the "line" by walking the track seems to be a waste of time, any professional will tell you the time invested up front will pay off in the long run.

In order to engineer software, a structure is required. Part of this structure is what I call professional programming practice. Professional programming practice consists of heuristics that aid us in developing software well. Professional programming practice won't tell you what algorithm or data structure is required to solve the problem. It will guide you in producing those algorithms and data structures effectively.

truth, beauty, goodness

Software engineering is about truth, beauty, goodness, and community in the context of developing systems to solve problems. The classical culture of Greece referred to truth, beauty, and goodness as the triumvirate.
[ref: http://www.labri.org/england/resources/05052008/AF02_Goodness_Beauty_3E64FE.pdf]
Dr. Gene Veith wrote, "Even aesthetic principles were equivalent to truth. Beauty was an objective quality in a work of art or of nature. Some works of art really were understood to be better than others. Not as just a matter of subjective personal taste, but as a matter of objective reality. "The true," "the good," and "the beautiful" constituted the three "absolutes." Truth, goodness, and beauty were objectively valid categories in the external universe."
[http://www.ligonier.org/learn/articles/consequences-truth/]

Each of these aspects is a symbol pointing to software. Software can be measured in terms of these attributes. Truth, beauty, and goodness are not only inescapable realities in human existence, they are also fundamental aspects of software engineering. As an aside, when Dr. Patrick Ferry

became president of Concordia University he identified truth, beauty, goodness, and community as his vision for the university.

It is easy to see that historic and current software does not possess the attributes of truth, beauty, and goodness. Software costs too much. Software takes too long to develop. Software is full of bugs. This "unholy trinity" of software problems needs to be addressed and the "software crisis" solved. Software engineering is a means to that end. Dr Fred Brooks first addressed these issues in the 1970s. His classic book, *The Mythical Man-month* shows some of the root causes of the software crisis and provides some insight into its solution.

For more than 25 years the Standish Group has tracked software project success and failure rates. The CHAOS Reports have been published since 1994 and are a snapshot of the state of the software development industry. The latest year publically available (2015) reportedly studied 50,000 projects around the world, ranging from tiny enhancements to massive systems re-engineering implementations. The report includes an enhanced definition of success looking at some additional factors which were not covered in previous surveys.

This table summarizes the outcomes of projects over a recent five year period using the new definition of success factors (on time, on budget with a satisfactory result).

MODERN RESOLUTION FOR ALL PROJECTS

	2011	2012	2013	2014	2015
SUCCESSFUL	29%	27%	31%	28%	29%
CHALLENGED	49%	56%	50%	55%	52%
FAILED	22%	17%	19%	17%	19%

The Modern Resolution (OnTime, OnBudget, with a satisfactory result) of all software projects from FY2011–2015 within the new CHAOS database. Please note that for the rest of this report CHAOS Resolution will refer to the Modern Resolution definition not the Traditional Resolution definition.

Successful means the project is completed on time and on budget, offering all features and functions as initially specified with a satisfactory result.

Challenged means the project is completed and operational but over budget and over the time estimate, and offers fewer features and functions than originally specified.

Failed means the project is cancelled at some point during the development cycle.

When software projects fail, they sometimes do so in a big way.

"U.S. Air Force pulls plug on ERP project after blowing through $1 billion
⌂
In November, reports emerged that the U.S. Air Force had decided to scrap a major ERP (enterprise resource planning) software project called the Expeditionary Combat Support System after it racked up $1 billion in expenses but failed to create "any significant military capability."
⌂
ECSS was supposed to replace more than 200 legacy systems. The project dated to 2005 and used Oracle software, but its ballooning costs clearly suggest that Air Force officials and systems contractor CSC conducted an overwhelming amount of additional custom coding and integration work.

⌂
An Air Force spokesman said the project would require another $1.1 billion just to complete one-fourth of the original scope, and that wouldn't be complete until 2020."
[http://www.computerworld.com/article/2493658/it-management/the-scariest-software-project-horror-stories-of-2012.html]

Other failures may be found here, http://calleam.com/WTPF/?tag=examples-of-failed-it-project

Software is different from hardware. Concepts related to hardware development and the lessons learned over many years may not apply to software. Unlike hardware, software is not a tangible entity. You can touch and feel a motherboard, but not software. Software is an informational entity. Hardware is manufactured and one motherboard may be different than another. After producing 100 motherboards on an assembly line, it might be that 3 are defective and can't be used. Software is cloned and there is no difference between the 100 copies that are produced for sale. Over time, the laws of physics dictate that the motherboard will wear out. Software does not wear out with use over time since it is an informational, and not a physical, entity.

In industry, software engineers perform two jobs: develop new software systems and maintain existing software systems. More software engineers are involved in maintaining software than creating new software. The maintenance aspect of software development includes both enhancing software and fixing bugs. Unfortunately, a large percentage of software engineering effort is expended in finding and fixing bugs.

Abstractly a computer is a tool for solving problems. When a user employs an application package, it is to solve a specific problem. Software engineers should also use tools to help them create systems. In addition to obvious tools such as compilers and development environments, there are a number of other tools that are classified as CASE, Computer-Aided Software Engineering, tools.

The Software Development Lifecycle, SDLC, is central to the process of software engineering. The SDLC is a framework for software development. The SDLC is a way to manage and monitor the software project throughout its lifetime. The SDLC is a conceptual tool that helps us develop software well. The SDLC helps us reach the goal of creating an acceptable system. An acceptable system is correct, usable, and developed on-time and within budget. There are many different specific implementations of the SDLC. Each has unique characteristics, terms, and phases. The odds of you using the exact SDLC we will follow in this text in the future are small. The point is to understand the value and purpose of a SDLC and not to be worried about the specific details. The actual SDLC used in different companies by different teams varies. There is no one correct software engineering process. The SDLC we will follow has 6 major phases: requirements, specification, design, instantiation, execution, and maintenance. While the terms and details vary widely, all SDLCs include aspects of these fundamental phases.

Dr. Locklair's* ZR-1 Corvette exiting the track at the Indianapolis Motor Speedway
(* technically my wife, Karen, owns the ZR-1, but I do get to drive and race it)

0 – Introduction to Software Engineering (part B)

Software Engineering is a response to a crisis. The software crisis, first identified in the 1960s, is demonstrated by software products that are incorrect, unusable, late, and over budget. Some have erroneously assumed that since there is an artistic aspect to software creation, a structure can't be applied to its development. The framework of software engineering is important because developing software is a complex activity. Rather than stifling creativity, the SDLC helps software engineers manage the creative process efficiently.

Software includes documentation. The dictionary entry on Software Engineering states, "Software products include the actual program source code and data structures, as well as the documents necessary to produce these, and documents and interface programs necessary to use them in the intended environment." This is surprising to coders and those who only program-in-the-small. Documentation is a vital aspect of software. Documentation includes everything that isn't directly executed by the hardware (source code). Documentation includes user documentation and user manuals. Documentation includes comments in the source code and following professional programming practice for literate programming (for example, using meaningful identifiers). Documentation includes notes relating to the requirements, specifications, design, instantiation, and maintenance of the software. Documentation includes recording insights from the SDLC process. To many coders, the idea of documentation seems wasteful and nonproductive. Why spend time documenting the system when the software *is* the system? Like most professional activities, documentation pays dividends in the long term. It is oftentimes difficult to remember why a certain approach was taken. Documenting the "whys" helps software engineers smoothly develop and maintain the system. Because software is developed by teams, documentation is necessary for effective team communication. Imagine a change being made to the spoiler of the race car to provide more down force. The pit crew does this but doesn't communicate what happened to the driver. This would cause problems during the race. Or, imagine if the crew chief notices that more camber (alignment issue) is needed on hilly race tracks. If this isn't documented, it won't be remembered for the next time the team races on a hilly track. Documentation is communication. Problems are solved via the communication of information. Documentation not only helps other people on the team, but it helps you to remember something several months from now.

The goal of software engineering is to produce acceptable products which consistently meet requirements. The requirements will dictate a number of specific attributes, but in general software must be correct, usable, cost-effective, and

developed on time. The SDLC helps software engineers produce acceptable products. Each phase of the SDLC begins with validation and ends with verification. Both validation and verification are designed to ensure the goals of the project are met. There are six phases in the SDLC we will follow: requirements, specifications, design, instantiation, execution, and maintenance. While there is an order to the phases, there is also some overlap. The creative process isn't always smooth and orderly. We will discover that we have to backtrack sometimes and jump ahead other times. The SDLC isn't meant to be a rigid straightjacket, but a guide. Some people incorrectly view the SDLC as a waterfall where you can only move forward and never go back. The dictionary entry on Software Engineering states, "No matter what overall development approach is taken, there are some fundamental activities that must take place to produce the software: requirements specification, design, implementation, and verification/validation. These activities are not necessarily disjoint in concern nor in time of occurrence."

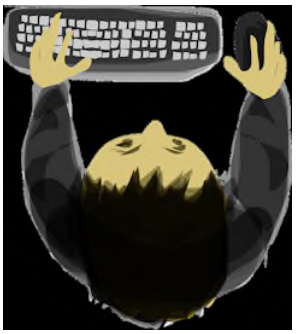
Software engineering is concerned about the quality of the product *and* the process. Professional techniques and development principles are encouraged by effective management of the project. One important principle in software engineering is abstraction. Abstraction allows us to focus on the big picture first and then investigate the details later. Abstraction allows us to hide details that aren't currently important. Abstraction is shown in many ways. The modular development of code, object oriented design and programming, code reuse (APIs and libraries), and the use of patterns are all derived from the fundamental computer science concept of abstraction.

In order to support the process of software development, CASE tools have been developed. Like software engineering in general, the use of CASE will seem like an unnecessary addition of complexity to novices. *Software Engineering Insider* commented, "Many studies have been done on the use of CASE tools, and the results point to their benefit but with the importance of correct use by a strong software developer. In other words, they do not ensure that anyone can write a good computer system, but they enable experienced software developers to do an even better job. So, if you are interested or already working in software engineering, consider extending your educational path to cover CASE tools, and you will likely become even more marketable as a software engineer."
[http://www.softwareengineerinsider.com/articles/case-tools.html#.VId5O53naUk]

CASE tools are useful in automating and documenting many phases of the SDLC. CASE tools are sometimes divided into upper CASE, focusing on the early stages of the SDLC, and lower CASE focusing on the latter stages of the SDLC. Diagramming software is one type of CASE tool. The ability to visualize ideas is powerful. Software such as Visio, SmartDraw, and Giffy allow software engineers to create UML, E-R, dataflow, and other diagrams. Project management tools are

useful for estimating time, cost, and resource allocation. Programming tools, such as an integrated development environment (IDE), are invaluable during the instantiation phase. Prototyping tools allow software engineers to quickly produce visual representations of the software in order to present ideas to users. Automated testing tools save time and effort during regression testing. The richness of CASE tools is amazing. The sad fact is that many programmers never take the time to learn the tools well.

The quality of the final software product is extremely important to a software engineer. The user will ultimately determine whether the project has been successful or not by how well the product meets its goals. Professional software engineers believe the process used in development can help ensure a quality product is delivered to the customer. There are a number of organizations that have investigated the characteristics of a quality process. Two of these organizations are the ISO and SEI. The International Standards Organization, ISO, has produced a number of standards. The ISO 9000 series deal with quality management. ISO 9001 provides detailed requirements for a quality management system. These guidelines focus on the process used, but not necessarily on the end product. ISO 9126 (superseded by ISO/IEC 25010:2011) deals specifically with characteristics of quality software in the areas of functionality, reliability, usability, efficiency, maintainability, and portability. The Software Engineering Institute, SEI, has researched ways to develop software that is correct, usable, on-time, and within budget. The SEI web site states,



"The SEI helps advance software engineering principles and practices and serves as a national resource in software engineering, computer security, and process improvement. The SEI works closely with defense and government organizations, industry, and academia to continually improve software-intensive systems. Its core purpose is to help organizations improve their software engineering capabilities and develop or acquire the right software, defect free, within budget and on time, every time."

"For four decades, the Software Engineering Institute (SEI) has been helping government and industry organizations to acquire, develop, operate, and sustain software systems that are innovative, affordable, enduring, and trustworthy. We serve the nation as a Federally Funded Research and Development Center (FFRDC) sponsored by the U.S. Department of Defense (DoD) and based at Carnegie Mellon University, a global research university annually rated among the best for its programs in computer science and engineering."

The SEI has developed the Personal Software Process (PSP) and Team Software Process (TSP) guidelines to aid software engineers in professional software development. The SEI pioneered the CMM (Capability Maturity Model) as a guide

to measuring the software development process. According to the CMMI web site, "Whether your business is developing high-tech systems, consumer software, or IT hardware, you want to ensure the highest quality product reaches your customer on time. The guidance provided by the free download of the CMMI for Development, "Version 1.3 (CMMI-DEV, V1.3) model uses management and engineering concepts to help you to achieve on-time delivery and high quality, especially if your product relies heavily on software.

CMMI-DEV guidance covers the lifecycles of products from conception through delivery and maintenance. CMMI-DEV best practices are flexible enough to apply to a variety of industries, yet stable and consistent enough to provide a benchmark against which your organization can measure and compare itself."

I. Engineering

What is engineering? The word comes from the Latin words *ingenium*, which roughly means clever and *ingeniare*, which roughly means devise. Engineers cleverly devise new structures. Engineering is the application of scientific principles to the creation of devices. Here are some dictionary definitions, "the branch of science and technology concerned with the design, building, and use of engines, machines, and structures;" "the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to people, and the design and manufacture of complex products <software engineering>;" "the art or science of making practical application of the knowledge of pure sciences, as physics or chemistry, as in the construction of engines, bridges, buildings, mines, ships, and chemical plants, and the action, work, or profession of an engineer, and skillful or artful contrivance; maneuvering."

What's the difference between science and engineering? Abstractly, science

discovers while engineering applies. Dr. Fred Brooks put it this way, "Perhaps the most pertinent distinction is that between scientific and engineering disciplines. That distinction lies not so much in the activities of the practitioners as in their *purposes*. A high-energy physicist may easily spend most of his time building his apparatus; a spacecraft engineer may easily spend most of his time studying the behavior of materials in vacuum. Nevertheless, the scientist *builds in order to study*; the engineer *studies in order to build*." Engineering is a synthetic discipline; engineers create and build things.

In many disciplines, there is a long history of successful engineering. Civil engineers and electrical engineers have a rich heritage to draw upon. Consider the

job of an automotive engineer. It is a good exercise to think about all the aspects of that job. How are new cars engineered?

Why is there a software crisis? One reason is related to the nature of software. If you asked a random person "on the street" to engineer a dam across the Missouri river, I doubt anyone would be willing to even try. On the other hand, if you asked people to write a software program to add numbers, I imagine that many people would be willing to try. It seems easy to create software. After only an hour of introduction, many people are able to hack out code. Even the president of the United States has "coded." Software is malleable, it isn't concrete. It seems easy to "dream up" and easy to modify. Managers without knowledge of computer science and information technology believe "making" software is easy. While anyone can code, engineering software is a difficult and complex task. It requires a structure and sound principles just as any other engineering discipline.

II. Software Engineering

Software engineering is the management of the entire software development process from the conception of an idea to the maintenance of its interaction. Software engineering applies sound computer science principles to the implementation of an acceptable software product. Software engineering changes an idea into reality by applying the software development lifecycle which consists of requirements, specification, design, instantiation, execution, and maintenance. The goal of software engineering is to solve a problem by creating an acceptable product. Software engineering can be described as *professional* problem solving.

There are two professional societies in computer science that provide helpful resources for software engineering. The ACM, Association for Computing Machinery, and the IEEE-CS, Institute for Electrical and Electronic Engineers – Computer Society. You should investigate both of these organizations.

There is a significant difference between the academic and production environments of software engineering. At school, student programmers are told what to do. In other words, the requirements and specifications are provided. Given those ingredients, the program is created. In the production world, software engineers must determine both the "what and the why" beforehand. While it may seem strange, oftentimes users don't really know what they need in order to solve the problem at hand. Before embarking on any development project, the software engineer must determine why there's a problem, what the problem is, and what is needed in order to solve the problem.

Coding is a small portion (percentage) of software engineering. Coding is important! Yet, it is usually less than 20% of the total project effort. Most coding is accomplished during the instantiation phase. Requirements, specifications, design, documentation, testing, maintenance, training, and a host of other activities make up software engineering.

A successful project produces an acceptable product. The process of software engineering was created to rid software of the unholy trinity of problems: over budget, behind schedule, and full of bugs. How do we know if we did the job right? If we met the user's expectations, then we were successful. The user will determine whether the product is a success or failure. The success of the product indicates the success of the project. A successful project is developed on time, within budget, and meets requirements, including correctness and usability.

Software engineering is the solution to the "software crisis." As computer science students we know that it is relatively easy to create small programs. You could create a program to add numbers, right? Therefore, we might assume that "size versus effort" is a linear relationship. A problem that is twice as complex will require about twice the effort, so we assume. This is not the case. Larger products (programs) are much, much harder to create. Programming-in-the-large is extremely difficult; therefore, we must plan well, communicate well, organize and evaluate in order to complete the task. Software engineering requires that we manage the process well in order to succeed.

Software engineering is the management of the software development process, including the following: expectations, computer science and technology, people (and their skills), time, cost, and other resources. There are as many social aspects of management as there are technical aspects. Managing expectations is ensuring that all parties involved understand clearly the scope of the project. If the user expects a touch or voice interface, but the product only allows keyboard input, expectations have not been met. It is the job of the software engineer to integrate computer science concepts and people skills in order to construct a system to solve problems.

The software development lifecycle (SDLC) is a way to manage the project by monitoring its progress at various stages from conception through retirement. The SDLC is a management tool. Each of the phases of the SDLC includes a number of "milestones," which mark our progress in software development.

As computer scientists and information technologists you understand the problem solving process. In its abstract form, problem solving consists of four activities: understand the problem, plan the solution, create the algorithm, and test the system. The actual problem solving process is more complex, of course. But at its

heart, problem solving involves analysis and synthesis. Analysis refers to breaking something down. When we investigate the problem and try to understand it, we are performing analysis. Some aspects of planning involve analysis. Planning also involves synthesis. Synthesis refers to building something up. Creating and testing are also synthetic activities. The SDLC incorporates both concepts of analysis and synthesis and expands upon our simplistic four-part problem solving model.

Software engineering, like all creative activities, incorporates an artistic component. While we want our products to be beautiful, we also want them to be true. Our artistic creations must also relate to, and fit in with, the real world. True creative activity involves a trinity of idea, implementation, and interaction. The idea is the mental picture of the problem and the proposed solution. The implementation is the work involved to change the idea into a tangible solution. The interaction is how people will use, and benefit from, the product.

In his letter to the Christians at Colossae, the Apostle Paul wrote, "Put on then, as God's chosen ones, holy and beloved, compassionate hearts, kindness, humility, meekness, and patience, bearing with one another and, if one has a complaint against another, forgiving each other; as the Lord has forgiven you, so you also must forgive. And above all these put on love, which binds everything together in perfect harmony. And let the peace of Christ rule in your hearts, to which indeed you were called in one body. And be thankful. Let the word of Christ dwell in you richly, teaching and admonishing one another in all wisdom, singing psalms and hymns and spiritual songs, with thankfulness in your hearts to God. And whatever you do, in word or deed, do everything in the name of the Lord Jesus, giving thanks to God the Father through him ... **Whatever you do, work heartily, as for the Lord and not for men**, knowing that from the Lord you will receive the inheritance as your reward. You are serving the Lord Christ." Colossians 3:12-17, 23-24 (ESV)

Dr. Locklair interacting with his ZR-1 Corvette on the track