CSC 478: Software Engineering Capstone

Module 1

Introduction to Software Engineering; Project Management

Objectives:

1. Understand what software is.
2. Understand what software engineering is and why it is important.
3. Understand what a software process is.
4. Understand some attributes of good software.
5. Understand some of the issues involved in selecting and retaining staff in a software development organization.
6. Understand factors that influence individual motivation and their implications for software project managers.
7. Understand key issues of team working, including team composition, team cohesiveness, team communications, and team organization.
8. Understand some of the basics of risk management, and know some of the risks that can occur during a software project.

Textbook:

1. Sommerville, chapters 1 and 22.
2. Web sections accompanying chapter 22:
    a. People Selection (https://iansommerville.com/software-engineering-book/static/web/people-selection/)
    b. Workspace (https://iansommerville.com/software-engineering-book/static/web/workspace/)

Assignment:

1. Answer one question in the discussion forum for Module 1, and respond to the post of at least one other student.
2. Complete the Online Resume Assignment for Module 1.

Quiz:

Take the quiz for Module 1.

1.1 Some Basic Questions

Chapter 1 talks about some of the most basic questions about software and software engineering. Software is pervasive. It is present not only in our personal computers, but also in our cars, phones, ATM machines, cash registers, etc. The list goes on and on. As both a society and as individuals, we are currently highly dependent on software for carrying out much of our day to day activities, and the degree to which software is pervading our lives is continually increasing.

But what is software? When people talk about software they are usually referring to computer programs or applications that can be used to serve some useful purpose, such as word processors, spreadsheets, graphic design, streaming video, etc. The list is almost endless. Defining what software is turns out to be not quite as simple as it may seem. Many definitions focus only on the executable program that runs when an application is launched. Other definitions go beyond the executable to add accessory files the executable may need to function. In the paragraphs that follow we will look at several aspects of software to arrive at the definition of software we will use in this course.

As you have probably noticed, most software applications require additional files beyond the executable file in order to function. These additional files can include configuration files, data files, audiovisual media files, language files, and user preference data files, to name a few. Some applications may require selected components from component libraries, which may be installed in their entirety even if only partially used. Without these files you may still be able to launch an application, but it may not function as intended, if at all. Thus, a functional application may extend beyond the coded instructions that were written and compiled to allow the application to perform its tasks.

Most software applications also usually come with documentation consisting of installation instructions and a user manual. Such documentation is not concerned with actually performing any of the tasks an application was built for, but may be necessary for end users to successfully use the application. Physical media such as CD-ROMs and DVD-ROMs have often been used to store and install applications, though there is a transition away from physical media towards network-based downloads and installations. Regardless of the nature of the storage medium, all software applications must reside on some type of physical media. Even if an application is not stored for reuse, it must at least reside in a computer's memory in order to function. User documentation has also transitioned away from paper hardcopies towards digital files, and are often incorporated directly into the applications themselves.

Most software applications are not written on impulse, with little or no consideration given to establishing user requirements, analysis and design, testing, or maintenance. Although end users may never see the documentation from the development process that was used to build an application, it can be argued that the application likely would not exist in its current form without this process documentation. Process documentation helps to ensure that the software that is built is the software that is desired, has been built correctly, and has been tested to ensure the software is as free of bugs as possible.

When defining the term, software, then, it is clear we must include the application itself, which consists of the program or programs that are loaded into memory and executed. We must also include any configuration files, data files, user preference files, language files, and other files the application references as it operates. All user documentation that was created as part of the application's development should also be included in the definition of software. Even though some users may never look at this documentation, others may be unable to install or use the software without these important documents.

Now we come to the question of whether to include the process documentation that is generated as software is developed. End users will never see this documentation, as it is never included in the package that is released to the public. The software itself will also never

need to have access to this documentation in order to function. Despite the transparency of the process documentation to the end users, we can still ask whether the process documentation is important to the software being able to function as intended and carry out all the tasks for which it was developed in the first place. I would argue that the process documentation does indeed play a vital role in allowing software to function as it was meant to do. Considerable effort may have been spent creating the set of requirements that define the scope of an application's function. Much time and effort is also spent designing how best to make the application meet all its requirements, then implementing all the required features in code. The effort spent during testing is vital to ensuring software is robust, and is released with as few bugs as possible. Periodic maintenance keeps the software up to date, corrects bugs that slipped through testing, and improves usability. Because of the importance of each of these process activities, any documentation that is generated as a result of the development process should be included in the definition of software, since it is difficult to argue the software would have functioned just as well had its requirements not been elicited, had no designs been developed, and had no testing been performed.

Now that we have settled on what defines software, we next ask, what is software engineering? Although many people envision software engineers as programmers who sit around hacking out code, there is a great deal more to building software than writing code. All software is written to solve some type of problem. Before a solution to the problem can be engineered, the problem must first be understood. Software engineering, then, could be defined as the analysis and understanding of some problem, followed by the synthesis of a software solution to that problem. A good software engineer must therefore also be a good problem solver.

We have mentioned software processes in the preceding paragraphs, but what is a software process? In the most basic sense, it is the procedure that is followed to build software. But building quality software is not a trivial endeavor — in fact it is often quite hard. Over the years software engineers have discovered that it helps to follow a well-defined and well organized procedure, as opposed to one that is ad hoc, to ensure quality and correctness. A good process involves planning, scheduling, documentation of activities, and coordination among team members, among other things. Although no two software systems are exactly the same, the activities involved in building different systems can be ordered so as to yield an optimal build time.

1.2 Software Engineering vs. Manufacturing

Software is not really "manufactured" like hardware such as a CPU chip, a hard disk, or any other physical object. Defects can be introduced into physical objects during the manufacturing process that render the object useless. For example, if the manufacturing process for making CPU chips results in chips with broken pins, the chips will be unusable. Physical objects are also subject to wear and tear. Over time, a hard disk will eventually wear out due to some problem - for example, the materials that make up the read/write head may erode to the point where data can no longer be reliably written to or read from the disk.

Software is not affected to the same degree by such physical defects. A corrupted copy of the software may be burned onto a CD-ROM because of a malfunction in writing to the media, but as long as the original copy of the software still exists somewhere another copy can be easily made. Once installed, software may also be rendered unusable because an essential file is corrupted or deleted, but the software can be easily reinstalled to correct the problem.

Although software does not "wear out," it does deteriorate. This is illustrated in Figure 1.1, which shows a comparison between the failure curves for hardware and software.

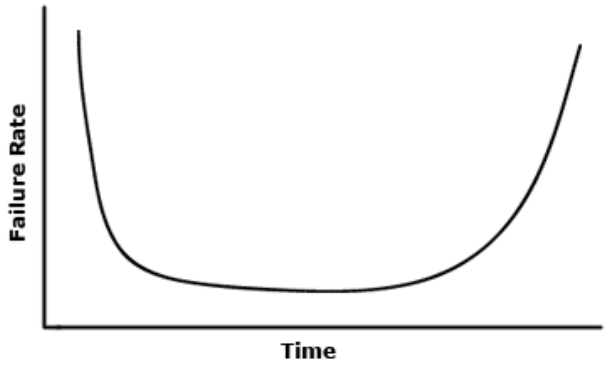

Figure 1.1a. Failure curve for hardware.

This is a typical "bathtub" failure curve for hardware. Initially there is high failure rate caused by defects. As defects are corrected, the failure rate declines, reaches a plateau where the hardware is more or less stable, then rises again as physical parts of the hardware wear out.

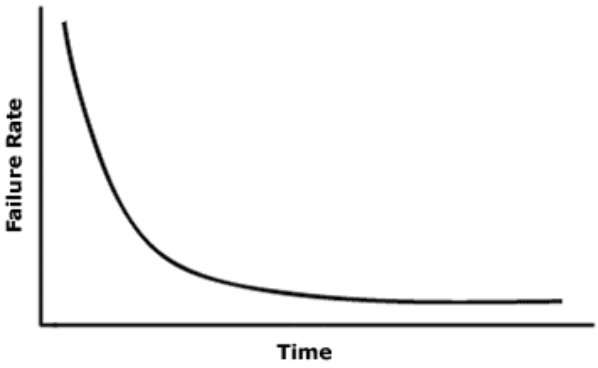

Figure 1.1b. Predicted failure curve for software.

Since software has no physical parts that can wear out, one might predict the failure curve to look something like this. There will still be a high initial failure rate, due to bugs. But as the bugs are corrected, the software stabilizes. Without physical parts to wear out, the failure rate does not increase again.

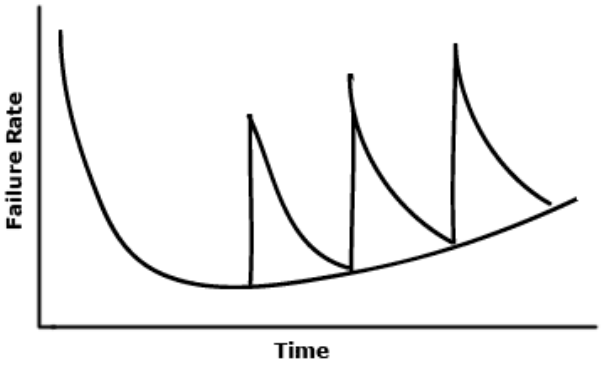

Figure 1.1c. Observed failure curve for software.

This is the type of failure curve that is typically observed for software. The spikes in the failure rate observed after the software has reached stability represent what happens when changes are introduced into the software. Each change potentially introduces new bugs, which cause the failure rate to rise. As bugs are fixed, the failure rate declines. However, as the software is continually changed, it becomes more complex and thus less resistant to failure. The software is not "wearing out", but it is deteriorating.

Figure 1.1. Failure curves for hardware vs. software.

1.3 Software Quality

What defines quality software? We'll cover this in more detail in a later module, but it's good to make a brief mention of software quality early on, just so the notion of what kinds of attributes can be involved in the definition of quality software. Many quality models have been proposed, most of which have a number of attributes ending in "-bility". One such model, McCall's Quality Model, proposes eleven attributes:

1. **Correctness** – without a doubt the software does precisely what it is supposed to do, and everything it is supposed to do

2. **Reliability** – the probability that, over time, the software will perform correctly

3. **Efficiency** – does it perform reasonably with respect to speed, resource consumption, etc.?

4. **Integrity** – how stable is it? How easily can it be penetrated by a malicious hacker?

5. **Usability** – how easy is it for the average end user to use?

6. **Maintainability** – how easy is it to upgrade and fix?

7. **Testability** – how easy is it to test?

8. **Flexibility** – how well can it handle unforeseen circumstances? If functionality must be changed, how easily can it be done?

9. **Portability** – will it operate in different environments (e.g., different operating systems)?

10. **Reusability** – can parts of the software be reused to build other software?

11. **Interoperability** – will it interact well with other software?

1.4 Personal Software vs. Public Software

Sommerville makes an important distinction between the development of software for personal use and the development of software that will be available to the public, with respect to adherence to quality standards and process models. When building software for personal use, you know that only you—and possibly a few other select individuals—will ever use the software. If the user interface is a bit clunky, or not all of the features work, or you have to employ awkward workarounds to get the software to do something, it's not really such a big deal. The people who will be using the software will most likely be aware of these problems, and depending on how extensively the software will be used, it may not make sense to try to produce a polished version. A lot of the effort that goes into software development has to do with the little details, after all.

However, if the software is going to be used by many people in a diverse end user population, it's necessary to try to make sure as few problems as possible make it into the final product. While some end users may be okay with employing tricky workarounds, some will not. If the software is to be marketed, excessive problems will reduce profits, lead to bad reviews, and could potentially cause people to lose their jobs. So for software that will be released to the general public (what Sommerville refers to as "generic" software), it is essential to follow a well-defined software development process. The extra time and effort spent will be well worth it.

1.5 Responsibilities of Software Engineers

Unfortunately, there have been numerous blunders created by humans that were ultimately found to be caused by software failures. (Visit the Risks Digest Website for lots of examples of past and current software blunders). A study conducted in 2002 by the National Institute of Standards and Technology (NIST) concluded that poor quality cost the software industry nearly $60 billion, with the end users bearing about two-thirds of that cost.

What responsibilities should software engineers bear when their software fails? Should they be held accountable for what happens as a result of their software being used, or should the responsibility fall on the users to know which software to buy and understand how to use it properly? In other engineering fields, engineers are required to be accountable for their actions. Each step in the engineering process must be signed off by an authorized engineer to certify that the step was completed properly. So, for example, if a building collapses because of a weak foundation, the engineer who signed off on that part of the process will be held responsible. It is assumed that the engineers take enough pride in their work that they will not certify anything they know to be substandard. It is difficult for the general public to trust in the safety of a building based on their limited knowledge of how buildings are built. It's much easier for people to trust licensed, responsible engineers who can assure them the building is safe because they know how to construct buildings, and by certifying a building is safe they are taking responsibility if it harms anyone.

If a bug is found in software, is it really the software engineer's fault? Building software is a very complex endeavor, and it is very difficult (if not impossible) to produce software that is fault-free. So if our word processor crashes unexpectedly once or twice a year, should we run out and sue the engineers who built it? Maybe the word processor crashed because of an operating system malfunction, not because the word processor itself is faulty. But what if human lives depend on software working reliably? In that case it's quite natural to expect the software to be free of faults. Such safety-critical software is required to fail no more than once in every 10^9 hours of operation; i.e., no more than one failure per 114,077 years. Of course, we can't really test the continuous operation of any software over such a time span, and even if we could, whatever software we were testing would certainly become obsolete 114,000 years down the road!

When it comes to building software, the picture is not as clear as to who should be held responsible. Computer software and hardware both are in a rapid state of flux, and it seems almost to require superhuman effort keep pace with all the changes. As long as everything works, everyone is happy, but when something goes wrong, everyone expects someone to be held accountable.

A great deal of controversy exists about whether people who build software should be allowed to call themselves "engineers." Practitioners in other fields of engineering are typically required to be certified or licensed periodically, and they usually belong to a union or guild that requires all its members to abide by an accepted code of ethics. It's not easy to become an engineer, and not just anyone can join the ranks. Software developers, on the other hand, are not required to be licensed or certified to do their jobs, and they are not required to possess as strong an education as, say mechanical engineers or electrical engineers. In fact, virtually anyone who wants to build software is free to try their hand at it, no matter how much training they have had. By cleverly wording the license agreement they can legally absolve themselves of any blame should the software fail. For example, some companies have avoided lawsuits by selling customers licenses to use their software, rather selling them the actual product.

In the mid-1990's the National Society of Professional Engineers (NSPE) sued in all states to prohibit anyone from using the term "Software Engineer" as a noun or as a field of employment. The NSPE argues that it has special rights over the use of the term "engineering." They claim that software engineers should have the same knowledge base as traditional engineers, with backgrounds in physics, chemistry, etc. The NSPE won this lawsuit in 48 states, although the judgment has been largely ignored. One difficulty is that no one can really agree on precisely what software engineers should know.

1.6 Project Management

Every project, even one that involves only a single person, will require some degree of management. When we speak about project management in the context of software engineering, though, we are generally referring to projects that involve more than one individual. A quick brainstorm reveals many attributes of a software project that will require managing, in order to complete the project correctly and efficiently:

1. staff availability
2. staff expertise and experience
3. the project's budget
4. time
5. resources (e.g., computers, software, office/cubicle/desk space, etc.)

Looking at the above list, it becomes apparent that the attributes listed are shared by all types of projects, even those that do not involve software engineering. Why is special attention given to project management in the field of software engineering? Sommerville discusses three reasons:

1. Software is relatively intangible compared to non-software projects.

2. The current project is often different from previously completed projects.

3. The processes used to develop software are variable from organization to organization.

Non-software projects such as building a house can easily be seen in their varying states of progress. One day only the foundation might be seen. A few days later the frames of the walls may be visible. Sometime later, the roof will be added, etc. It's often fairly easy to tell how far along a project is simply by looking at what has been done so far. With software projects, it isn't that simple. A software project consists almost entirely of digital documents (source files, database files, library components, word processing documents, etc.). The only tangible objects a software project ever really has are installation media (e.g., optical disc) and printed documentation, and nowadays even those are no longer produced in a physical format. Most software is now typically installed from a server or downloaded directly to a user's computer, from where it is installed; physical storage media are now used only for backups. During development, it is often difficult to determine how close a project is to completion, because there are few readily visible cues. It is common for a software project to be split into several smaller sub-projects, each of which is completed by a different software engineering team, which may not even be located nearby geographically. Also, more effort must be put into determining the completion status of a software project, since many digital documents must be read and analyzed, unlike a house building project where one could simply look at the house.

It is a generally known fact that as people perform a task repeatedly, they will tend to become faster and more efficient at completing that task. Increased familiarity with a problem through repetition is one primary reason for this. Another is the fact that as a task is performed, problems that are encountered will usually be solved, and will either not be present in later repetitions of the task, or will be solved more efficiently. Continuing with the house building project as an example, if a team of contractors are hired to build a series of identical houses in a neighborhood, the houses that are built later will likely be built in less time and with greater efficiency than the houses built early on. Software projects, on the other hand, very often tend to be different enough from previously completed projects such that repetition does not play such a large role in improving the efficiency of development. Although software engineers can make use of reusable components and concepts such as design patterns to increase efficiency, the point is that the same project is never really developed twice.

While we have made great strides in improving the processes by which software is engineered, software engineering as a practice still suffers from insufficient standardization. The assembly line developed for the construction of the Ford Model T in the early 1900's had a drastic influence on the way many tangible products are manufactured. The whole notion of products in various states of completion travelling along a conveyor belt, with employees stationed along the way adding one component at a time, can be applied to a whole range of different physical products. In software engineering, certain processes can be applied to some projects, but not all projects. The two processes that seem to be most generally applicable are incremental development, where each iteration of the development cycle produces a product with increased functionality; and stepwise refinement, where a complete product is developed, but is refined during later iterations to improve efficiency, correct minor details, etc. But the process used by a given organization tends to be the one with which the development team has been most productive with. The problem is that while this process may have worked well for past projects, it may not work for the current project, and the development team is forced to retool and use a different process.

I propose adding a fourth reason to Sommerville's list: a software project is a highly complex piece of engineering, and management is essential to rooting out all the details of a problem. A software engineering team is required to elicit a set of requirements from the customer that adequately defines the functionality of the software to be built. It is very easy for some requirements to be missed during this process, or for some of them to be changed as development progresses. What looks like a complete analysis of a problem at the end of requirements elicitation will typically not be complete at all. Assumptions made during the design phase may result in serious faults when the project is implemented. Less than thorough testing may fail to reveal these faults, and the released software may be responsible for serious losses in money, assets, and/or lives. If you're not convinced of the complexity of software, try developing a complete (i.e., all details are covered) specification for a relatively simple project such as the game of Yahtzee, or some other game of your choice, then try to prove the specification is complete.

1.7 Team Structures

Most of the programming projects you've done in your college career have probably been solo projects that have been relatively small (say, a few hundred lines of code). Most of the projects tackled in industry, however, are too large and complex for a single individual to complete in a reasonable period of time. Such projects are built by one or more teams of individuals.

Soon, I will divide the class into teams for the group project for this course. Many people cringe at the thought of having to work as part of a team, especially as part of a class. Probably the most common criticisms of group projects in a classroom situation are lack of communication and laziness — often there is a "slacker" who contributes little to the project.

In order for a team to work together it must have a defined structure. Teams with an ad-hoc structure (i.e., you make things up as you go along) are rarely successful. Each team member needs to have a defined role and a clear understanding of what the boundaries of that role are. There are three general types of team structure:

Democratic Decentralized

In a democratic decentralized structure, all project decisions are made democratically. There is no defined team leader. The advantage here is that everyone has an equal role on the team, so individual egos are least likely to be bruised with this structure. The disadvantage is that progress may be very, very slow if the team members can't come to agreement on issues.

Democratic Centralized

In this structure all team members still get to provide input on all project decisions, but there is a defined team leader who has the ultimate say on all matters. The presence of a team leader helps to ensure progress will not stall due to indecision. The same person may be the leader throughout the project, or the role can be shared.

Controlled Centralized

Here, there is also a defined team leader, but in this case the leader makes all project decisions. Other team members play subordinate roles. They may still offer suggestions, but the project is directed by the leader. The chief programmer team is an example of a controlled centralized structure. This type of structure is good when success depends on realizing one person's vision of what the project should be. However, it can cause problems if the other team members are not satisfied with their subordinate roles.

1.8 The Surgical Team

In the 1970's, Fred Brooks was the project manager in charge of the OS/360 operating system for the IBM 360 computer. He proposed the following team structure, which he dubbed "The Surgical Team":

The Surgeon (chief programmer)

The surgeon designs, writes, codes, & tests the software, and writes all the documentation.

The Copilot

The copilot knows as much about the project as the surgeon, and can take over for surgeon if disaster strikes. The copilot can propose ideas for the project, but the surgeon doesn't have to use them.

The Administrator

The administrator deals with things like personnel, raises, space, hardware & software acquisition, etc. The surgeon is too busy building the entire system to personally deal with all the details of these issues. However, the surgeon does have the final say on all decisions made by the administrator.

The Editor

Although the surgeon writes all the documentation, essentially the documentation the surgeon provides is only a first draft. The editor is responsible for doing any necessary reworking, cleaning up, organizing, and such to ensure the final documentation is readable and understandable. The editor also oversees the production of all written manuals.

Two Secretaries – one for administrator, one for editor

It is assumed that neither the administrator nor the editor will be able to physically perform every task necessary for them to complete their jobs. Therefore, each has a secretary to help out with any tasks that can be delegated.

The Program Clerk

The program clerk maintains all technical records, as well as the code repository (for reusable components & project versions). This individual is thus responsible for maintaining and archiving all versions of a project.

The Toolsmith

The toolsmith builds auxiliary tools, as needed, that aren't specifically part of the project. Examples of such tools would be installer programs, compilers, code formatters, and tools for organizing data that must be handled for the software.

The Tester

The tester is responsible for creating all test cases, along with all scaffolding code (test harnesses, etc.) necessary for performing the tests. The surgeon, however, is the one who actually executes all the tests.

The Language Lawyer

Every programming language has its own set of tricks that can be employed to allow tricky or obscure things to be done with it. The language lawyer is a super-guru for whatever programming language is being used. If it is at all possible to accomplish something with a language, the language lawyer will know how to do it.

Keep in mind, at the time Brooks was leading the creation of the OS/360 word processors were not common, and there were no scanners, laser printers, or other devices we commonly use today. It was not uncommon for the documentation for a project to take up an entire shelf, so making a copy of it was a Herculean task.

1.9 A Note Regarding Motivation

One of the biggest obstacles a software development team faces is lack of motivation in one or more members of the team. The sequence of events that ensues is fairly common. A team member loses motivation to work on the assigned project, for one reason or another. This lack of motivation almost invariably leads to a loss in productivity, which causes delays in the project. The delays can be handled in a variety of ways, but quite often those who are still motivated are forced to pick up the slack, which may lead to their own loss of motivation. The explanation of what motivates people is complex, not completely understood, and varies from person to person. Sommerville broaches this subject in chapter 22 when he talks about Maslow's model of motivation. To summarize, and put a light twist on Maslow's model, each person has certain needs that must be met in order for them to be motivated. The pyramid scheme Maslow uses (shown in chapter 22) probably applies pretty well to most people, but the numbers and priorities of these needs will likely vary from person to person.

Unfortunately, lack of motivation can be particularly problematic in an academic course — such as this one — that involves a group project. Not every student will be motivated to strive for 'A'-level performance in a group project. This could be due to laziness, burnout, mental and/or physical conditions, and a host of other possible reasons. Students who work full-time jobs and have families will often be forced by necessity to place those responsibilities first, which can lower motivation to work on an academic project that provides no income. Sometimes what appears as a lack of motivation is not lack of motivation at all. Lack of communication within a team can easily misinterpreted as a lack of motivation if a team member becomes confused and waits too long to communicate this confusion to the rest of the team.

1.10 Risk Management

Before talking about risk management, we need to clarify precisely what a risk is. A **risk** is an event that will cause some kind of loss if it happens. The associated loss is known as the **risk impact**. Every risk has a probability of occurring. The impact of a risk can also be mitigated to some degree. Risks in software engineering can be organized into three broad categories:

1. project risks
2. product risks
3. business risks

Project risks involve some aspect of the development process, but not the software being developed. A programmer may quit, someone may have a family emergency, a storm could fry sensitive equipment, etc. Product risks affect the software that is being built. For example, if the chosen programming language does not come packaged with any useful library components such as data structures, these will have to be developed from scratch. Business risks relate to the organization that is developing the software, and/or the customers who will buy the software. An obvious example of a business risk is a competing organization that is developing a similar software product. Another good example is the possibility that the business logic will change on the customer's end. If a bank has contracted for software that will manage the accounts of its customers, and late in the development process decides to change the way accounts are to be managed, the changed requirements will result in a delayed release of the software. Each of the above categories of risk can be further broken down into whatever resolution is necessary for effective risk management.

The first goal of risk management is to identify all the risks associated with the project. In so doing, a risk table is usually constructed which contains for each risk the identity of the risk, along with information about the 3 major characteristics of a risk:

1. the probability the risk will occur
2. the impact the risk, if it does occur
3. the degree to which the risk can be mitigated

Once the risk table is built, the risks are typically categorized according to probability and impact. They are then prioritized, with high-probability/high-impact risks at the top and low-probability/low-impact risks at the bottom. The project manager must then decide which risks can be managed, given the available resources. A cutoff line is established, and all risks above the cutoff line are managed, while those risks below the cutoff line are not.

Larger organizations may have the resources to establish a Risk Mitigation, Monitoring, and Management (RMMM) plan. The sole purpose of the RMMM plan is to manage risks. There may even be staff members dedicated to nothing more than risk management. The three "M's" are defined as follows:

1. **Mitigation** — avoidance is the best policy, so avoid the risk if possible

2. **Monitoring** — factors that could precipitate a risk are closely monitored, and appropriate actions are taken to minimize the probability the risk will occur

3. **Management** — essentially damage control; a risk has occurred, despite efforts to prevent it, so now the effect of the risk's impact must be minimized

References

1. Brooks, F. P. *The mythical man-month*, Addison Wesley Longman, Inc., 1995.