Module 12: Systems Engineering; Dependable Systems

Objectives:

1. Understand what is meant by a socio-technical system.
2. Understand what emergent properties are, and how they can arise in software.
3. Know how organizational processes, laws, etc., can affect the way software is designed and used.
4. Know what a legacy system is, and the problems associated with them.
5. Know what the five orders of ignorance are.
6. Understand what a critical system is, and what types of critical systems there are.
7. Know the four primary dimensions of system dependability.
8. Know the difference between the following terms: error, fault, and failure.
9. Understand the following approaches to improve system reliability: fault avoidance, fault detection and removal, and fault tolerance. Also understand the equivalents of these approaches with respect to ensuring safety and security.

Assigned Reading:

Sommerville, chapters 19 & 10

Assignment:

Participate in the group discussion for Module 12. You must make at least 3 posts, total, to receive full credit.

Quiz:

Take the quiz for Module 12.

Lecture:

Faults, Errors, and Failures

Before continuing any further, I should clarify a few terms that we have seen in the past, and that we will see again: *fault*, *error*, and *failure*. People often get confused about these terms and use them interchangeably. However, they all describe different things. A **failure** is when a software system does not behave as it is supposed to behave. A failure is not always equivalent to a crash, but a crash is certainly one manifestation of a failure. An example of a minor failure would be if you saved a text file, but the last line was omitted when the file was saved.

Failures are always the result of errors. An **error** is an unexpected system state that will likely lead to a failure, though the failure may not be immediate. An example of an error is an incorrect value returned by some computation, and could have arisen for many reasons. Incompatible units could have led to the error (this was the reason for the infamous Mars probe crash several years ago). An imprecise value caused by inaccurate floating point operations is another example of an error. A non-mathematical example would be storage of an incorrect value as a result of a human user entering invalid data.

A **fault** is some characteristic of a system that could potentially lead to an error. I say potentially, because a fault will only cause an error if the code that contains the fault is executed. A fault can exist for a long time in a system and never lead to an error because no one uses the feature whose code contains the fault. An example of a fault is a method that returns an integer when a floating point value is required. Another example would be lack of user input validation. Faults are more commonly known as **bugs**.

Thus, these three entities happen only in a particular sequence: fault --> error --> failure. Another way of distinguishing them is that faults, if there are any, are always present, whether the software is executing or not. Errors and failures can only occur at runtime. There are certain situations where an error can occur that was not caused by a fault; a hardware "hiccup" is an example of such a situation, although one could argue that the hiccup resulted in a transient fault that could lead to an error. It is impossible to proceed from a fault directly to a failure; there must be an erroneous system state, or error, in order for a failure to occur.

It is also important to mention that faults do not always result in errors, and errors do not always result in failures. A software fault can only cause an error if the code containing the fault is executed, and even then the fault may only cause an erroneous system state for certain inputs. Depending on how the software is used, a fault may lie dormant for a long time before it is discovered. Similarly, an error must somehow lead to incorrect or unexpected behavior in order for it to cause a failure. If an erroneous system state occurs, but is not

involved in subsequent program execution, it will not lead to a failure. Errors may also exist only transiently. An erroneous value may be overwritten before it has a chance to cause a failure, for example, or there may be validation mechanisms that detect the erroneous state and take appropriate actions before a failure can occur.

The Human Factor

All software that is used has an impact on society. Some software, such as software that will be widely used or used to support critical objectives of an organization, will obviously have a much greater impact than other software. Nowadays it is common for an organization's livelihood to be dependent upon a particular software system. Without it, the organization will not survive. The internet has transformed the way we shop, learn about current events, do research, and communicate with each other.

But software itself is nothing more than a tool used by humans to accomplish some objective. It is unaware of what it is doing, and knows nothing of the role it plays in the bigger picture. Humans are prone to making mistakes, and they can be unpredictable. There is no guarantee that a particular person who is responsible for operating a computer system will always provide valid input to the system, or that they will only use the system for the purpose for which it was originally designed. Often people do this out of ignorance, but they may also do it because they aren't able to acquire the proper software, so they have to make do with what they have. Sometimes people will abuse software for malicious purposes.

In some cases there are laws and regulations governing how certain software may be used. These laws can exist at the levels of federal, state, or local governments or they can be imposed by an organization. For example, many organizations restrict the use of email for work-related purposes only. At the federal level, there are laws against deploying computer viruses and worms. In many cases the only practical solution to controlling how software is used is to have acceptable use policies that people are expected to follow.

The term, **socio-technical system**, emphasizes that software has both a technical component (the software itself and the hardware and software that supports it) and a sociological component (the people that use the software or who are affected by the software). One of the key points Sommerville makes in chapter 10 is that software engineers need to be aware of how the software they build will fit into society, and what effect it will have when it is used.

The sociological component of a socio-technical system is primarily what leads to existence of legacy systems—systems that perform their required functions, but which are built on obsolete technology. Business policies, laws, regulations, etc., can change more quickly than software can be modified to accommodate those

changes. As we saw from Module 1, changes to existing software systems can eventually lead to instabilities. At some point the only sensible thing to do is build a new system. Unfortunately, not every company may be able to afford to pay for a new system to be built. Such massive changes also often require hardware upgrades, which only add to the cost. It may also become clear that user interface to the system needs to be changed to increase efficiency. However, the end users may balk at any attempt to change the interface, since they are used to doing things the old way and don't want to be burdened with having to learn how to use the new system. Surprisingly enough, people will often prefer an old, inefficient system that frequently breaks down simply because they are more comfortable with it. They are familiar with the old system and how to deal with its problems, and they fear having to learn a whole new system.

The Sociotechnical System Stack

Sommerville decomposes a sociotechnical system into a 7-layer stack, as shown in Figure 10.1 in the text, and reproduced below. He states that most interactions tend to occur between neighboring layers, and each layer tends to hide the details of the layer directly below it. But, some interactions can jump layers, or affect more than one layer. Suppose a law is passed that requires certain types of data must not only be encrypted according to a particularly strict standard, but the data must also be able to be processed within a specific time window. This legislative change happens at the societal layer, but causes a cascade that ripples through potentially all the layers below. Business policies may need to be changed at the organization level; the business processes that govern how the system is used may also need to be modified; the application system itself will need to be modified to add the required encryption capabilities; software that resides between the application system and the data (e.g., a database management system) might need to be altered to accommodate changes made to the application system; the time window for data processing may require an upgrade from 32-bit hardware to 64-bit hardware; and as a result of the hardware change, a new operating system may need to be installed. Alternatively, if we assume the system was originally designed to foresee such a change in the law, the impact of the new law may only extend down as far as the business processes layer.

What does all of this mean? It means that when you are designing a software application, you should take into consideration the entire system with which the application will interact, and not just the application itself. In the event your application experiences a failure, the failure should ideally be confined to your application, and not extend into the system beyond. This is why creating a context model is so important, especially for complex systems. Without a context model to indicate where the boundaries of your application lie and how your application interacts with other parts of the overall system, there is no way you can accurately say what will happen if your application should experience a failure.

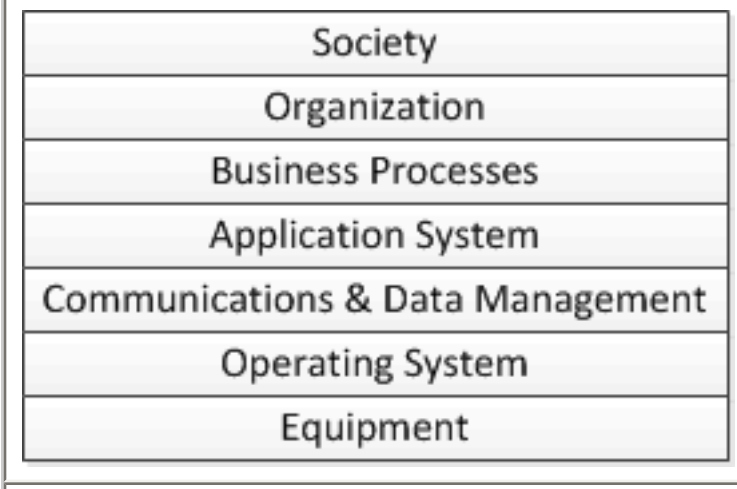

Figure 12.1. The sociotechnical stack depicted by Sommerville in Figure 10.1 of the text.

Emergent Properties

Emergent properties are properties that arise out of the complexity of a system, and essentially make a system larger than the sum of its parts. Emergent properties tend to arise because certain combinations of activities, or the presence of certain combinations of interactions, can have unexpected effects. Ant colonies are excellent examples for illustrating emergence. (Those of you who took my Data Structures course may remember how the ant colony simulation project demonstrated emergence). Individually, each particular type of ant (forager, queen, soldier, etc.) performs a specialized task and by itself cannot explain the existence of the colony. When all the types of ants work in combination, however, a viable colony (an emergent property) can result.

Emergent properties in software tend to cause more harm than good. The main reason for this is that it isn't always possible during development to accurately know what other software may be running on the end user's machine, so it often isn't practical to take steps to avoid specific problems that arise from concurrently running programs. As an example of a bad emergent property, suppose you have two independent programs running, each of which tends to hog system resources. When each program is executed alone, the system remains stable. When both programs are running concurrently, however, they may end up creating a deadlock situation where the computer freezes, or they may cause such extensive use of virtual memory that the hard drive "thrashes". Unexpected emergent properties in software systems are like mutations in a living cell. More often than not, the result will be harmful.

That's not to say that all emergent properties are bad. Consider the Microsoft Office suite. Each tool (Word, Excel, PowerPoint, etc.) alone provides a useful set

of features. But if combinations of the tools are installed, you now have additional capabilities such as being able to copy and paste an Excel table directly into a Word document. Good emergent properties usually arise because developers make a special effort to enable their software to interact with some other software. Companies can strike bargains in which technology is shared to enable emergent properties that will benefit both companies.

With respect to software, Sommerville breaks down emergent properties into two broad categories: 1) functional and 2) non-functional, and the decomposition closely mirrors that used to categorize software requirements. Functional emergent properties are related to the functionality explicitly described in the requirements. Some functionality may not become available until certain components have been integrated together, thus this functionality emerges from the functionalities of the individual components. Non-functional emergent properties cover all the properties that emerge from a system that are **not** related to specific functions the software is supposed to perform. Good examples of non-functional emergent properties include:

1. performance
2. reliability
3. dependability
4. security

For example, an application may perform well within all specified time requirements during testing, but may fall short when it is integrated into its actual environment. Network bandwidth slowdowns, hardware conflicts, and delays caused by other running applications and processes can all conspire to reduce an application's performance. Unexpected interactions between an application and other components in the system can reduce both reliability and dependability. Finally, holes in the security of other system components may allow intrusions that are capable of overcoming the security measures engineered into the application. With respect to reliability, we are accustomed to recognizing that failures can occur due to software faults, or that hardware failures can lead to unexpected conditions that can lead to software failures. However, if we recognize a software system as a sociotechnical system, we now also need to take into account the possibility that a failure can result from user error. If a user enters invalid data, or tries to use software in a way it was not intended to be used, it may lead to a software failure.

The Five Orders of Ignorance

For any given project, there are going to be things you know and things you don't know. It's easy to classify the things you know, since you are aware of those things. But what about the things you don't know? Is it possible to categorize what you do not know? It may sound surprising, but the answer is, yes. Lack of knowledge is *ignorance*, not to be confused with terms like foolishness or stupidity. There are five orders of ignorance, described below in the context of software engineering projects. Keep in mind these orders of ignorance can be abstracted to cover any topic.

0th order ignorance

0th order ignorance is a complete lack of ignorance with respect to some project. Lack of ignorance can be demonstrated by building the right system the first time (not counting things like syntax errors caused by poor typing skills). 0th order ignorance is for the most part impossible to achieve, although individuals that have accumulated a great deal of experience can sometimes approach a level between 0th order and 1st order.

1st order ignorance

1st order ignorance is when you don't know something, but you are aware of what it is that you don't know. For instance, suppose you are working on a module involved with image processing, and you know that you need to use a Fourier transform, but you don't know how to do a Fourier transform.

2nd order ignorance

2nd order ignorance is when you don't know something, but you don't know what it is that you don't know (if you did, it would only be 1st order ignorance). However, with 2nd order ignorance you do know how to find out what it is that you don't know. Suppose you are working on the image processing module mentioned above, but in this case you do not know that a Fourier transform must be used to accomplish your goal. After several failed attempts to come up with an algorithm that works, it dawns on you that there must be something about the problem, of which you are ignorant. You don't know what that "something" is, but you do know of several books you can refer to, and maybe you have a colleague experienced in image processing who you can consult. After consulting these references, you discover your algorithm needs a Fourier transform and you then proceed to learn how to incorporate a Fourier transform into your algorithm.

3rd order ignorance

With 3rd order ignorance, there is something you don't know—but you don't know what it is that you don't know—and furthermore you have no idea how to find out what it is that you don't know. This level of ignorance is commonly found among those who attempt to solve a problem that is way beyond their experience and knowledge. Continuing with the image processing example, an individual with 3rd order ignorance would be unaware that the algorithm they are trying to write needs to use a Fourier transform, and they would also be unaware of how to go about finding this information. Cutting edge research projects are also good examples where 3rd order ignorance applies. In these types of projects, investigators are typically required to become pioneers in their fields, and innovate new methods for discovering the unknown.

4th order ignorance

I like to call this meta-ignorance, because it essentially means you are ignorant about the 5 orders of ignorance. Before reading this lecture, you may have had 4th order ignorance, but for at least a short time after reading this lecture (depending on how long you remember this) you will not have 4th order ignorance.

I'm not sure who originally came up with the five orders of ignorance, but the article by Armour [1] is a good reference.

Dependability & Security

Sommerville defines software dependability as a measure of how much confidence a user has that the software will behave as expected, and not fail during normal use. Dependability cannot easily be defined numerically, because although the number and frequency of failures can be quantified, trust varies from individual to individual, and is thus subjective in nature. The ubiquitous advice to maintain backup copies of your files is a testament to the acknowledgement that software can only be trusted to operate without failure for so long, so users should adapt their work habits to accommodate software failures. Software dependability has four primary aspects:

1. Availability

   This is the probability that, at any given time, the software will be running and will be capable of *attempting to do* what is asked of it.

2. Reliability

   This is the probability that, at any given time, the software will *correctly* perform what is asked of it.

3. Safety

   This is the probability that the software will in some way cause harm to people or its environment.

4. Security

   This is the probability that the software will be able to resist intrusion of any type.

In addition to these four main aspects, dependability also consists of a number of other, auxiliary properties, including:

1. Repairability

   This is a measure of how easily and how quickly a system can be repaired; i.e., brought back online, following a failure.

2. Maintainability

   Since according to Lehman's laws, software must change over time in response to changes in its environment, in order to remain useful, maintainability measures how easily and how quickly changes can be accomplished.

3. Survivability

   This is a measure of how well a system can continue operation, or at least avoid failures, during emergency situations, such as network attacks, power failures, or failures in other systems.

4. Error Tolerance

   This is a measure of how well a system is capable of behaving as expected even in the presence of errors; e.g., user input error, users incorrectly using the system and causing errors, errors propagated by other, connected systems.

Dealing with Faults to Improve Reliability

As was mentioned earlier in the course, software can be extremely complex. Ambiguities or gaps in the requirements can lead to design flaws, which can in turn be translated into code faults. Code faults may also be generated independently of any design flaws. The purpose of testing is to discover faults, but in practice it is impossible to guarantee that software is fault-free, even if it passes all the tests that are used during the testing phase. Exhaustive testing would need to be performed in order to make such a guarantee, and exhaustive testing is impractical for all but the most trivial programs. We must accept, then, that even rigorously tested software may contain faults when it is released. Fortunately, there are some things we can do to mitigate the impact of those faults:

1. Fault Avoidance

   We can first try to minimize the number of faults by employing certain programming techniques, such as structured programming; avoiding the use of pointers and GOTO statements; ensure that all variables are assigned meaningful default values upon declaration; reuse code that has already been thoroughly tested; and following a code formatting standard that promotes readability—to name just a few. The reasoning behind this approach is that many faults are a direct result of poor programming practices, so by eliminating those practices or replacing them with better ones, the number of faults will be reduced.

2. Fault Detection and Removal

   As I mentioned at the start of this section, software is tested in order to find faults, so testing is one primary means of detecting faults. Other techniques include formal code reviews, as well as design reviews, since it is possible to have code that functions correctly according to the design, but if the design contains a flaw, so might the code.

3. Fault Tolerance

   At this stage, it is assumed that as many faults have been avoided or detected as possible, but there still may be faults present. The goal of fault tolerance is to try to ensure the software will still operate as expected even when a fault produces an error. Since many faults are caused by human input (e.g., entering invalid data), including mechanisms to validate all input before it is processed can greatly reduce the frequency of faults caused in this way. This may involve forcing a user to input dates strictly in the format MM-DD-YYYY, for example. As another example, if numeric input is required, validation checks can be performed to catch non-numeric input, possibly directing the user to re-enter the input. For situations that are internal to the system, validation checks can be included to ensure certain variables never contain values outside the valid range. This is sometimes called "bulletproofing" the code, when it is done extensively throughout a program. Another method of providing fault tolerance is to use redundancy at one or more levels. For example, two

different algorithms that perform the same function might be used, and the results compared. If the results are the same, it is assumed there are no errors; a difference in the results indicates an error has occurred somewhere. System-wide redundancy may also be used, where there are redundant copies of the entire software system. Such extensive use of redundancy is usually reserved for critical systems, however.

Dealing with Faults to Ensure Safety

A similar, three-pronged, approach can also be used to ensure software does not cause harm to people, other living things, or the environment. With respect to safety, the terminology is slightly different, but the pattern is basically the same:

1. Hazard Avoidance

   The software is designed to reduce the number and frequency of situations where harm may occur. Software itself does not directly cause harm, it can only cause harm through its interactions with physical devices, so hazard avoidance may entail designing the system to prevent, or at least strictly control, human interaction with those devices.

2. Hazard Detection and Removal

   The system is designed to detect hazardous situations and take appropriate actions to either neutralize the hazard or signal for anyone nearby to leave the area.

3. Damage Limitation

   Although system faults may be able to be tolerated, the option of tolerance really doesn't exist for hazards. Should a hazard occur, the goal here is to try to mitigate the damage caused by the hazard. For example, the system may detect that a fire has started, and in response trigger for the area where the fire occurred to be sealed off in order to contain the fire. The system may also simultaneously activate sprinkler systems, issue an evacuation order, and even notify local emergency authorities.

Dealing with Faults to Ensure Security

As with reliability and safety, we see the same tripartite approach, with the key terms again changed to follow common software security jargon:

1. Vulnerability Avoidance

   Any form of protection that attempts to prevent intrusion can be considered part of vulnerability avoidance. This includes the use of passwords, encryption, secure network connections, biometric authentication, etc.

2. Attack Detection and Neutralization

   Antivirus software is a prime example of this approach. Custom software can also be written to monitor network traffic, system usage patterns, dictionary-type attacks on authentication modules, etc., and take various actions if suspicious activity is detected. In an extreme situation, the system could be brought offline or shut down in order to prevent a potential intrusion.

3. Exposure Limitation and Recovery

   Should an intrusion attempt be successful despite the system's attempts to neutralize it, the system may be able to limit the damage by immediately moving or deleting sensitive data to prevent it from being accessed. A successful intrusion is analogous to a power failure in that there must be some recovery mechanism that allows the system to resume normal operation once the threat has passed. Part of such a mechanism would be the ability to maintain backup copies of data, purge the current database (since false data may have been added as part of the attack), and restore the database with the data from the backup. Another approach that may work in some circumstances is to maintain "decoy" data, which are bogus data created and stored to look like genuine data. It is possible that an intruder may be fooled by the decoy data, buying time for the system, or a system operator, to intervene.

Critical Systems

In order to determine whether or not a software system is a critical system, you have to think about what can happen if the software fails. If failure of the software can result in something very bad happening, such as injury, death, serious economic loss, or some other serious consequence, the software is a critical system. Software that controls the flight of an aircraft, for example, is a critical system because if it fails, and the aircraft crashes, people could die. If you are an organization that does the majority of its business through e-commerce, and the servers controlling your online transactions fail, you could lose a lot of money.

The effects of software failure can be relative, though. For example, suppose you are making a grocery list using your favorite word processor, and in the middle of typing "milk" the software unexpectedly quits, and you hadn't yet saved the file. This is obviously not very critical, even though it does cause inconvenience. Now, suppose you are using the word processor to write your Master's thesis. Assume you have learned your lesson from the grocery list failure, and have been diligently saving your file periodically. However, the next time you try to open the file to continue working on it, you discover that the software somehow corrupted the file the last time you saved it, and half of your thesis is now missing. This is more critical than losing the grocery list because a Master's thesis is clearly more important than a list of food items, but it is not likely to cause injury, death, or economic loss. Finally, suppose the same word processor is used in a hospital by a worker who is updating a list of all patients and the medications they are to receive. Unbeknownst to the worker, there is a fault in the software that causes the last line of text to not be saved unless it is followed by a carriage return, and a patient dies because they did not receive their medication, since their name was omitted from the list. This time, there is no question about the severity of the loss. If the word processor is an integral part of the hospital's routine, it is a critical system (specifically, it would be a safety-critical system).

How do you define the dependability of a critical system? If lives could be lost if the software fails, it may be tempting to require that the software should never fail. Anyone with reasonable experience with computers will soon realize, though, that this is impossible to guarantee. One definition does state that the software should not fail more than once in 10^9 continuous hours of operation. But 10^9 continuous hours of operation is approximately 114,000 years! There are several problems with this definition, which I leave to one of the discussion questions for this module.

Case Study: SCADA Systems and the Power Grid

This is old news now, but many of you may recall the enormous power blackouts that occurred in the eastern U.S. in August, 2003. They were the largest blackouts in U.S. history, affecting about 50 million people. Over 100 electric plants were shut down (22 of which were nuclear plants). Ten major airports had to shut down, cancelling over 700 flights. In the New York subway system, over 350,000 people were trapped [2]. There was much speculation about why the blackouts occurred. It is believed by one source that a broken alarm at an Ohio utility company, First Energy, may have initiated the cascade of events that caused the blackouts.

Power grids these days are set up such that if one location in the grid needs additional power, this power can be drawn from other parts of the grid that can spare the power. Thus, it is possible for everyone (hopefully) to have enough power without having to depend on the local utility companies to supply it all. A software system known as SCADA (Supervisory Control And Data Acquisition) gathers real-time data from locations throughout the area serviced by the grid, and allows technicians to monitor power usage and control how the power is rerouted. Some power rerouting can occur automatically, as well. In the case of the aforementioned First Energy company, one report I read stated that the SCADA system noticed a need to reroute power to a particular part of the grid. The necessary power was rerouted, but unbeknownst to anyone, there was a problem at the location where the power was sent. The rerouted power was immediately channeled back to its source, and the resulting feedback caused a massive system overload, thus setting into motion a cascade of failures that ultimately resulted in huge blackouts.

According to analysts, it is predicted that we should experience such a large power outage about once every 35 years. Interestingly, the last power outage of comparable magnitude occurred in 1965, so according to those predictions we were due for another one. Due to the complexity of the power grid and the demands made upon it, no one can predict with any certainty when a massive blackout will occur, or where it will occur. Current models show that even slight fluctuations in small areas of the grid can precipitate massive instabilities. According to chaos theory, blackouts appear to be normal and natural byproducts of the power grid system. Another hypothesis states that failures are inevitable and unpreventable in a complex system, since we, as humans, cannot fully understand such a system well enough to devise a fault-free solution. Proponents of the hypothesis argue that since prevention of failures is impossible, it's pointless to waste time trying to prevent failures and instead focus on trying to survive them.

References

1. Armour, P.G. The five orders of ignorance. *Communications of the ACM 43*, 10 (2000), 17-20.

2. Fernandez, J.D. and Fernandez, A.E. SCADA systems: vulnerabilities and remediation. *Journal of Computing Sciences in Colleges 20*, 4 (2005), 160-168.