Pledge: *I pledge my honor that I have abided by the Stevens Honor System.* - **Eric Altenburg**

1.6: As software becomes more pervasive, risks to the public (due to faulty programs) become an increasingly significant concern. Develop a doomsday by realistic scenario in which the failure of a computer program could do great harm, either economic or human.

A realistic doomsday scenario in which the failure of a computer program could do great harm would be applicable to the Coronavirus outbreaks that have been happening recently. Software is at the center of diagnosing illnesses and monitoring medical procedures, and if one of these software systems were to malfunction, then there can be serious repercussions such as injury and even death. Especially today, the ability to accurately diagnose an individual is extremely important to prevent the continuing spread of this disease. Another real-world example would be if open heart surgery were occurring and then the machine that is used for monitoring the patient's current health were to malfunction, then the patient's health and well being would be at risk. The two aforementioned scenarios are only the tip of the ice berg when it comes to large side effects of software failing. As software continually becomes more intertwined with our day-to-day lives, it opens up more opportunities for the ways in which it can fail. What is normally accomplished now through mechanical means will likely be replaced with software in the future, and this opens up doors for more doomsday scenarios.

2.8: Is it possible to combine process models? If so, provide an example.

Yes, it is possible to combine process models, in fact, some software development departments and companies do not use a traditional process model. Instead, they end up using a proprietary process model which is just a combination of other more traditional process models that better suit their work flow/products. One such example is the evolutionary process model which is a combination of the iterative and incremental approach. Over time, the team will incrementally create a more complete version of software (more so than the last) and for each of thees incremental builds, a complete cycle of activities are completed. Another example of a combined process model is the spiral model. It is here where a combination of iterative and sequential linear models are combined with the use of the waterfall method with an emphasis on risk analysis. Finally, the last example of a combined process is the incremental process model where one or more waterfall models are used. This model produces a series of releases that provide more functionality for the customer and these builds are individually designed, tested, and delivered at specific deadlines.

2.9: What are the advantages and disadvantages to developing software in which quality is "good enough"? That is, what happens when we emphasize development speed over product quality.

When considering the disadvantages of developing software whose quality is considered to be "good enough," it should be mentioned that software is heavily reliant on other software to function. When new software is developed, it could be absolutely perfect, however, if it is reliant on other functions or programs, then this "perfect" piece of software could become faulty. Therefore, the foundation must be very strong; no weak links. If an individual develops software that is "good enough" and it is largely independent, then using discretion for this "good enough" measurement is okay, but as mentioned above, if other programs rely on it, then precautions should be taken. In summary, what might be considered to be "good enough" at the time will slowly turn into "not good enough" resulting in lost time, money, and other human resources such as motivation or productivity trying to fix the old code.

With that begin said, there are a few scenarios where developing software that is "good enough" is acceptable and even advantageous. When discussing the topic of research, a final product is not necessary and usually a working concept is only required, therefore, no additional software will be developed that will rely on the "good enough" software. The same applies to start-ups and other experimental products as most of the time, one is trying to show that a specific problem can be solved with the use of software.

3.2: Describe agility (for software projects) in your own words.

The agility software process is designed to deliver a product in a fast manner, and for the software to be delivered per the customers' specifications. If the customer decides to change their requirements, then the developers will be able to adapt and accept the challenges the new requirements may bring. The overall goals of agility mean that the software must be kept as simplistic as possible in that, only the requirements are to be met; nothing extra which helps with scope creep. Agile software development is also more growth oriented as well meaning that although it will have to adapt to new challenges and requirements, it will do so incrementally to avoid drastically disrupting the pace of the project and to make certain that any progress made will always be forward and no backwards. This also allows for the cost of the new changes to be minimal as everything is controlled and introduced in incremental steps. Some methodologies include extreme programming, dynamic systems development method, lean development, and feature driven development. In summary, it uses iterations, incremental delivery, and continuous feedback to hone in and fully refine the final product for the consumer.

5.1: Based on your personal observations of people who are excellent software developers, name three personality traits that appear to be common among them.

Some of the top software engineers generally share the same personality traits as they help in development and managing their time. The best software developers must have the level of technical skills required to be able to fully understand the legacy system in which they are working with, or possibly a client's system. They then figure out how to make improvements to the system, or propose a new one that will prove to be better than the previous. This individual will also have to figure out how to implement the new system with the older one. Another personality trait is their morality and ethical sense. This is crucial as they may have access to personal details of a company that, if put in the wrong hands, can ruin said company or business. Specifically nowadays with AI, the developer must be aware of how they are coding up a program or application and how it can potentially be used. The final personality trait these engineers share are their orientation toward other individuals. In most cases, software developers will be working in a team or managing a team, and to do so efficiently they must be able to properly work with individuals and express their thoughts effectively. For those working in upper management, they must properly delegate individuals to do tasks while being fair to the other team members in terms of work distribution and existing personal issues.

6.6: Of the eight core principles that guide process (discussed in Section 6.1.1), what do you believe is more important?

The eight core principles that guide processes—be agile, focus on quality at every step, be ready to adapt, build an effective team, establish mechanisms for communication and coordination, manage change, assess risk, and create work products that provide value for others—are all important under their own respective circumstances, however, I feel as though that building an effective team should be placed above the rest for a few reasons. When developing software, the individuals someone is paired with for a project are the people they will be interacting with the most; therefore, it is crucial that these individuals get along. Mutual respect and trust are a requirement among team members as the entire base to a team is build on this, without it, little to no work will get done due to differing views or the way processes need to be handled. No one can trust each other to assign tasks and get work done, and this can quickly derail projects. Each member must also be diverse and have their own respective skill-sets because too many assertive members can result in lots of clashing opinions, and no work will get done. On the other hand, too many passive individuals can lead to nothing being done or potentially the wrong thing being done; no one will speak up or suggest a different approach to an issue. It is for these reasons that picking a team with a diverse skill set and the potential for lots of respect and trust to be built be considered for the utmost priority over the other seven core principles.

7.1: Why is it that many software developers don't pay enough attention to requirements engineering? Are there ever circumstances where you can skip it?

One of the most difficult tasks a software developer will encounter is understanding the requirements as often times there is a severe separation between the customer's requirements and the actual construction of the product. Therefore, for this reason, most software developers will end up ignoring the requirements engineering and instead, will take an abbreviated approach. Though in reality, every task must be given a dedicated amount of time to make sure the requirements engineering is performed thoroughly. This is done so because requirements engineering serves as the bridge between the customer's requirements and the product design.

Also, requirements generally do not stay consistent throughout the life of the software development process, therefore, since they change so often, developers end up ignoring them and build a generic prototype that allow for easy changes to be made per the new requirements. This especially pertains to a project in which there are a large amount of requirements, the developers will most likely ignore them due to the changes that will most likely ensue as mentioned above. However, if the software to be developed is for something where there are only a few requirements, then the developers will take their time and thoroughly delve into requirements engineering.

7.5a: Develop a complete use case for making a withdrawal from an ATM.

Use Case: User withdraws from an ATM

1. The ATM is on and in a state where it is ready to perform a withdrawal.
2. The user inserts their debit card through the card reader.
3. User uses the key pad to input their secure pin.
4. If the pin is rejected, then the user can re-input their pin up to 3 times.
5. If the pin is accepted, then the user is presented with a menu where they can perform a withdrawal, deposit, transfer funds between accounts, etc. (the options depend on the machine).
6. User selects withdrawal.
7. User selects or specifies a valid amount of funds to withdraw from their account. If the amount is invalid or the denomination of the funds the user is requesting is not available (i.e. the ATM ran out of $20 bills), then a message is displayed to inform the user.
8. After receiving their funds, the user must remove their card from the card reader, take the dispensed funds, and either select if they want a receipt or not.
9. The ATM logs out of the user's bank account when the card is ejected.

8.1: Is it possible to begin coding immediately after a requirements model has been created? Explain your answer, and then argue the counterpoint.

If a software developer were to immediately begin coding after a requirements model has been created, then they miss out on creating a coherent analysis model which serves as the backbone to the overall project. By not creating an analysis model, the design and interfaces will not be fully thought through. While yes, it is possible to begin coding immediately after a requirements model has been created, later on, problems might arise as the architectural design, interfaces, and the global data structure will not be fully implemented. This will end up consumer more resources than previously thought, and has the possibility to severely set back a project

While doing the aforementioned might lead to problems down the road, there is a benefit to doing so. By beginning the development process early on, the skeleton of the code can be created allowing for a faster prototype coming about under circumstances where there might not be much structure needed or when there are few requirements. If errors arise in the code based on structure, interfaces, and design, and if the prototype is developed correctly and in a modular manner, then the fixing/patching process will be much smoother.

8.10: How does a sequence diagram differ from a state diagram? How are they similar?

The sequence and state diagrams are both crucial aspects of a requirements document as they outline the overall functionality of a program. However, they each have their own differences and similarities depending on how they are being interpreted. For the differences, in terms of how the system behaves, the sequence diagram will give a detailed description of how the classes will move from behavioral state to the next. The state diagram, however, will not have any information about classes. Also, the state diagram will explain how a singular class will change states based on external factors. What the sequence diagram does different is show the behavior of the software with respect to time. The two diagrams are more different than they are similar, but one similarity they do share is that they are both used to model the behavior of the software, just done so in different manners.