Max Shi
CS 347 B
Professor Peyrovian
I pledge my honor that I have abided by the Stevens Honor System.

Homework 1

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

Security breaches remain one of the most economically damaging situations that can happen in society today. If a large company like Google had a security breach, and the encryption on the passwords were cracked, hackers could gain access to millions of accounts through shared passwords. While this situation isn't very likely, there are instances where large companies have been exposed to have large security vulnerabilities, such as the time it was uncovered that Facebook was storing some passwords in plaintext. However, the repercussions of this scenario would be massive. With the sheer amount of people who possess at least one Google account, these accounts could share passwords with other accounts from other emails to bank accounts to corporate logins, all of which, in the wrong hands, could do a lot of economical damage. Furthermore, the fact that email accounts with Google are often used as password recovery accounts for other websites would mean that a cracked password from Google could allow hackers to gain access to many other accounts without a shared password, causing a chain reaction that could lead to huge economic damage to an individual, even to the point of which their identity is stolen. Thus, a huge password breach could have a large economic and human impact.

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

It is absolutely possible to combine process models. Under most software development models, each model will have some sort of communication, planning, development, testing, and deployment phases. The differences lie in how the processes try to anticipate delays and changes in the project. For example, agile process models anticipate less concrete requirements and changes in project specifications so that a team can better adapt to them, while the waterfall model assumes well-defined requirements to create a very efficient and straightforward process to get the job done while minimizing extra steps such as prototyping. Combining different models serves to strengthen a model’s response to different problems encountered in the process. Furthermore, combined models already serve as models used in the real world.

For example, the incremental model is a software development process model that aims to develop software in small portions until all requirements of the customer were met. The most crucial requirements are targeted first, and then all other portions are developed separately and combined together to form the final product meeting all requirements. The prototyping model, on the other hand, tries to create working prototypes of the final software, allowing the customer to try the software, and using feedback to improve the software to meet customer demands. Between these two models, the incremental model is good at easy testing through making only small changes and getting the customer involved in the process, while the prototype model is especially good at getting the customer involved with hands-on versions of the project with prototypes. These two are combined to create the spiral process model, which begins with developing a prototype of the most crucial requirements and then continues to add on the rest of the requirements in the form of prototypes. Eventually, when all the necessary remaining requirements are enumerated, the software development team aims for a final spiral in the process to combine all elements of the project together. In this sense, it has combined the incremental model’s idea of implementing specific requirements at a time and combining at the end with the prototyping model’s idea of delivering prototypes to get the customer involved. Thus, the spiral model can be seen as a combination of these two models, that combine the incremental model’s strength of easy testing through small changes with the hands-on experiences for the customer from the prototyping model.

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

Even though developing software to be “good enough” seems to be a bad idea, it does have its advantages. In the pursuit of “good enough”, the development team tends to stay more focused on the core goals of the project, rather than adding extraneous “nice to have” features into the final project. As a result, the development team creates a working product meeting requirements in a shorter amount of time, which also means less development costs. Another advantage to consider with a shorter development time is how relevant the final product becomes. Software and technological development is very competitive, and if a company is able to deliver a product faster than competitors, they will gain an advantage in the market. If a project takes too long to deliver, it might not be relevant to the customer anymore, or the customer may be dissatisfied with how long the project took to develop. Developing software to be perfect takes a lot of time and effort, and may lead to missed deadlines, extra requirements, and, in a worst-case scenario, a scrapping of the entire project due to runaway costs.

However, there are obvious disadvantages to adopting a “good enough” mindset. In the pursuit of a product that is good enough to meet requirements, little thought might be put into the long run expandability of the product. In an extreme case, a “good enough” mindset might not be able to adapt to changes in requirements during development due to a more severe lack of expandability. Finally, developing software to be “good enough” may entail a lack of exhaustive testing to the final product, possibly allowing some bugs through to the customer. If the software is mission-critical software, letting bugs get through to the customer could have disastrous results. Thus, while there are clear advantages to developing software to be “good enough”, it is more important to strike a balance between the two extremes, to ensure a quality product is delivered on time.

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

Agility for software involves one main goal: to be able to adapt to changing requirements quickly while still meeting all the requirements and needs of the customer. There are many software development processes to accomplish this, but to achieve this goal, agility generally requires a couple essential concepts embedded within the software process. These concepts involve evolutionary prototyping, constant communication with the customer, and cross-functional teams that develop different parts of the software. These few concepts ensure that the software development team can adapt to changes in environments. Constant communication with the customer allows the software development team to ensure that they have a constant grasp of the most up-to-date set of requirements from the customer. Evolutionary prototyping ensures that the customer knows what the development team is creating so that they know what the final product might look like. These two concepts form a two-way channel of communication so that both the development team and the customer meet expectations and requirements. Furthermore, cross-functional teams ensure that different teams are responsible for different parts of the software, allowing for compartmentalization of software functions and a better ability to adapt to the changing requirements. In my mind, all agile software development processes have some manifestation of these concepts in order to achieve that final goal of being able to adapt to the changing requirements of the customer.

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

Based on my personal observations, people who are excellent software developers tend to have a few common traits between them. Out of those traits, the three most important ones I observe are that they are effective communicators, abstract thinkers, and detail-oriented people.

One of the more overlooked traits in the average software developer is their ability to communicate. While software development may seem to be less people-focused, typing on a computer all day, software development, in reality, requires a lot of communication with different kinds of people. A software developer may need to explain their code in detail individually to their peers, then turn around and give a presentation to their team about what they have been working on, then explain to a project manager or their boss about their project in more abstract terms, and then discuss the project requirements with the customer of the project they are working on. In each of these scenarios, a software developer needs to change many aspects of their communication such as the level of abstraction regarding their code, or the manner and tone of their speech when talking to co-workers compared to their boss or the customer. In order to accomplish all these aspects of software development well, it is apparent that a good software developer needs strong communication skills.

While this was mentioned while talking about strong communication, software developers need to be masters of abstract thinking, as it applies not only to communicating, but the technical aspect of the code as well. Regarding communication, a software developer needs to be able to talk about the intricacies of their code to their colleagues working directly on the code base, but also needs to explain their higher-level goals at team and project meetings. Furthermore, the software developer needs to be able to talk about the overall goals of the project to bosses or customers. In all of these examples, there are different levels of abstraction regarding the same topic. In order to be an effective communicator, abstract thinking is a necessity. However, this also applies to the technical part of the job. Abstract thinking allows developers to better apply their knowledge over multiple situations. Being able to boil code down to their techniques and applying those techniques over different applications makes a good developer great. They do not have to be taught new techniques for every language and project that they take on; they can simply take old knowledge, make the necessary adaptations, and begin applying their expertise again.

Finally, strong software developers are detail-oriented people. They are meticulous in their pursuit of what exactly needs to be done and how exactly to do it. A very important side effect of this is minimizing vagueness when it comes to all aspects of the project, something which, if left in, could spell disaster for a project. In their pursuit of details, they outline every aspect of the project, know how to execute their portion of the project, and do not miss errors or oversights that could impact the project later down the development cycle. These traits, again, cycle back to what I mentioned before as well. To ensure that their attention to detail does not go to waste, they need to be able to communicate the details that they notice, and need to be able to use different levels of abstraction to communicate to different kinds of people.

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

I believe “Principle #8: Create work products that provide value for others” is the most important principle out of the eight principles. This principle boils down all the other principles in the process, and creates a common context for all the other goals. By saying “others,” we emphasize that we create useful products for not only the customer, but coworkers as well. This means principles like “build an effective team”, “assess risk”, and “establish mechanisms for communication and coordination” are all encompassed under the general idea of creating work products (communication channels, teams) that provide value for coworkers. Continuing on, principles like “manage change”, “be agile”, and “focus on quality at every step” contributes to the idea of creating a final product that provides value for the customer. In this case, the work product is the final product, and the others are the customers. For example, software development should be agile in order to create a final product that meets the expectations of the customer. Software development should manage change in order to prevent large hiccups in the development process. Thus, principle 8 is the most important, because it provides a big picture encapsulation for the rest of the principles that govern the software development process toward the goal of creating a process that creates products to benefit all parties.

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?

Many software developers do not pay enough attention to requirements engineering due to the problems that can happen during the creation of the requirements. Requirements engineering requires constant and clear communication between the software development team and the customer, which can sometimes create conflict between the two parties. For example, the customer might insist on a requirement that the software development team believes is technically infeasible with the amount of resources (money, time) invested in the project. In this case, a lack of understanding between the two parties can stall the project, or create unreasonable demands.

Another aspect to consider is the frustration of requirements engineering. As mentioned in class, developing a requirements document can take many cycles and iterations of drafting a document, bringing it into a meeting, and getting the document verbally torn to shreds as stakeholders discuss the necessity of each requirement. Thus, this lack of positive feedback can make software developers feel like nothing is getting done, and that they are wasting their time writing requirement documents instead of developing a working prototype. This is especially apparent for more simple projects, where a software developer may be more inclined to form ideas about the final project in their head and build that themselves rather than enumerating and submitting documents for the requirements engineering process.

However, there are very few circumstances where requirements engineering can be skipped. There are processes to which this process can be made easier, such as giving the software development team more liberty in the creation of the final product, but requirements engineering should still be done to ensure the entirety of the team are on the same page. The situation where requirements engineering can be skipped boils down to the situation where a small team with good coherence and communication are creating a small-scale project, such that constant communication itself can act as a substitute for the coherence that the requirements engineering process creates.

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

Use Case: Withdrawal from an ATM
Actor: Customer
Goal: Withdraw cash from bank account using the ATM.
Preconditions: Customer possesses a valid ATM card and has a bank account with enough money for the withdrawal. The ATM must be operational.
Trigger: User inserts ATM card into ATM.

1. The ATM asks the customer for their bank PIN.
2. The customer inputs the valid PIN.
3. The ATM prompts the customer to choose their selected action from possible actions (withdrawal, deposit, etc.)
4. The customer chooses to withdraw money.
5. The ATM prompts the customer to input the amount of money they would like to withdraw from preset options or a custom amount.
6. The customer inputs their desired amount.
7. The ATM ensures that the customer has enough money in their bank account.
8. The ATM ensures that the customer has not exceeded their withdrawal limit.

9. The ATM verifies there is enough cash stored inside the ATM.
10. The ATM charges the customer’s bank account.
11. The ATM puts the cash into the slot for the customer to take.
12. The customer takes the cash.
13. The ATM prompts the customer whether or not they want a receipt.
14. The customer selects their desired option.
15. The ATM prints or does not print a receipt depending on the customer’s choice.
16. The ATM returns the customer’s ATM card.
17. The customer takes their ATM card.
18. The use case ends and the ATM is ready for another transaction.

Exceptions

1. The customer inputs the incorrect PIN.
2. The customer does not have enough money in their bank account.
3. The customer’s withdrawal request exceeds their limit.
4. The ATM does not have enough money stored to fulfill the customer’s request.
5. The ATM does not have enough paper to print the receipt.

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

It is possible to begin coding immediately after a requirements model has been created. Ensuring there has been good communication between the customer and the software development team during the whole process of creating the requirements document, it is unlikely that the final product will require a complete overhaul of the entire document and all the work done on the project if the coding were to begin early. There are likely to be certain non-negotiable core components and functionalities that can be worked on immediately after the requirements model has been created, and if a team were to begin immediately after the model was created, they should try to focus on these aspects of the project first. However, while the coding begins, parts of the team should still work on verifying the created requirements model with the customer to ensure that the model is accurate before trying to deliver a prototype. As long as the coding remains open to modification and future changes, it is fine to begin coding before the requirements model has been fully verified.

However, the counterpoint exists where failing to verify the requirements model with the customer can lead to an overhauling of the entire project and, in the worst case, wasted work time and money. It is possible that the created requirements model does not meet the customer’s vision at all, and requires a complete overhaul to ensure the customer is satisfied with the final product. Furthermore, it is also possible that the project is of such a huge scale that the risk involved with beginning the project early is too great to begin coding. These situations also tie into a previous answer where I discussed working to make software that is “good enough.” This prioritization of speed over quality and accuracy may lead to less expandability in the future, or even bugs developed in this code that did not get checked as the specs changed from the initial requirements document. Thus, while it is not forbidden to start developing code right from the completion of the requirements model, there are things to consider when deciding whether or not to begin writing the code.

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

A sequence diagram typically is limited to one use case, and details how each of the different classes in a system interact with each other to complete the behavior of a use case. While the sequence diagram can depict one or multiple paths through a use case, it typically has a clear beginning and end for the use case, never returning to a previous element in the diagram. In contrast, a state diagram depicts all the states a system can be in during the normal use of the system, which can encapsulate multiple use cases. The state diagram is allowed to go back to previously entered states as necessary in the behavior of the system. Furthermore, they are drawn very differently, with state diagrams being a finite automata, while sequence diagrams being arrows drawn between column-like structures representing the different classes in the software. In essence, the state diagram represents all possible execution states the system can be in, while the sequence diagram defines the behavior between classes within a state or number of states in the state diagram. In this sense, they are similar due to both being interaction diagrams representing aspects of the execution of the software, albeit in differing amounts of detail and abstraction within the software itself.