Ryan Filgas

CS300

Chris Gilmore

Midterm 1

1. Software is defined by computer programs and their associated documentation. Software is designed to accomplish specific tasks for a customer or a general market. Software engineering is an engineering discipline concerned with all aspects of software production including, but not limited to requirements gathering, design, production, documentation, project management, and the development of tools and methods to support software production.
2. Validation: Are we building the right product according to the initial requirements laid out? Verification: Are we building the project right to satisfy customer needs? (Somerville, 228). Verification checks that the program meets its functional and non-functional requirements. Validation ensures that the software meets customer expectations. Beyond functional requirements validation aims to satisfy that the software does what the customer expects it to. Sometimes what is set forth in the requirements at the end of the day isn't precisely what the customer needs, and validation hopes to satisfy this.
3. The four process activities for requirements elicitation and analysis are "requirements discovery and understanding", "requirements classification and organization", "requirements prioritization and negotiation", and "requirements documentation" (Somerville, 113).
    - Requirements discovery and understanding are important because the client talks about projects in their own language with specific knowledge of their own work and domain. Without having experience in the customers domain, a requirements engineer may not fully understand their needs. The goal of requirements discovery and understanding is not only to gather requirements, but to bridge the gap between an engineer's understanding of the project

requirements., and a client's understanding of the project requirements. Sometimes a client will need extra communication and questioning to really dial in what the client needs and how it should work as they might not fully understand themselves either what they need, or what can be accomplished with the technology, time, and budget given.

- Requirements classification and organization takes the raw information so to speak from requirements discovery, and organizes it into related sections that are easier to digest or understand. If requirements gathering is the first draft of a paper, requirements classification and organization is the second. It helps organize the requirements so they can more easily be understood.
- Requirements prioritization and negotiation prioritizes which requirements are the most important, and thereby resolves requirement conflicts. This may require negotiation and compromise between different stakeholders whose requirements are at odds with each other.
- Requirements documentation is the process of documenting requirements after the above steps are completed. Requirements may be revised at different stages of development and may be written informally in shared spaces initially.

4. The large advantages of prototyping come with change anticipation. A prototyping system allows engineers to refine requirements in specific areas before committing to high software production costs (Somerville, 61). It also helps with elicitation and validation of system requirements in smaller increments despite potentially changing requirements. Prototyping allows users to see how well the system supports their work, and it allows engineers to hone in on system requirements that are missing certain aspects or may need changes (Somerville 62). The net effect of this is that change proposals after delivery may be reduced. This development method is generally effective for user interfaces or games for example. In addition, prototyping allows for improved design quality, improved maintainability, reduced development effort, improved system usability, and a closer match to users' needs (Slides CH2, Slide 38).

Prototyping, while appropriate in some cases does have some disadvantages. One disadvantage is that prototype testers may not be typical of the systems users. In addition, users may not be trained properly to use the prototype given time constraints (Somerville, 63). If objectives aren't made clear in the prototyping process, a prototype may not be an effective way to refine requirements. In addition to this prototyping isn't appropriate for some situations. Critical systems for example shouldn't generally have prototyping because the consequences may be life threatening, they need to be fully specified before production. Prototyping may also be inconvenient for a working business. They don't want to use part of a system as they have work to do. In this case they need a fully developed product ready for a quick software change. In this case there should be launch ready with minimal errors as it can cause headache and cost to the business.

5. The five software development activities common to all software processes are software specification, software design, software implementation, software validation, and software evolution (Slides CH 2, Slide 3)
    - Specification is the "what" of the system. What requirements must the system fulfill for the client or user?
    - Design is how the system will be built. This is almost its own step, but it's rare for a system to be built in a low-level way such that the engineer has no choices and just implements design. Therefore, design and implementation tend to overlap.
    - Software implementation is the physical creation / coding of the software. Depending on the development process, this may overlap with other software development activities.
    - Validation, which includes testing makes sure that the software does what the customer wants as laid out in the requirements and does it well. This is not to be confused with verification.

- Evolution is changing the system in response to changing customer needs. This might include changing requirements or adding new ones. (CH2, Slide 3 and lecture.)

6. Unit Tests. Based on the lecture and content in the book. Unit tests are usually written by the software developer and run before committing to a repository. Based on lecture, this process can also include the potential for small integration tests for components local to the developers own code. As I wasn't sure what this question was specifically looking for, I've also gathered the answer from another source as double verification.

   - As found on amazon web services description of continuous integration (Amazon Web Services):

     "With continuous integration, developers frequently commit to a shared repository using a version control system such as Git. Prior to each commit, developers may choose to run ***local unit tests*** on their code as an extra verification layer before integrating."

   - From my interpretation the following happens after changes are committed by the developer and picked up by the "build agent". Because of this system tests happen only after the commit, and of course validation happens in a later stage of development.

     Quote from class: "when you check something into source control, a build agent picks up all those changes, usually after a short wait which we'll talk about, builds the system, runs unit tests, and then deploys the system if possible and then runs system tests, usually … building, running unit tests is common across pretty much every integration build"

7. In modified waterfall the steps are straight forward. Requirements are followed in order by system and software design, implementation and unit testing, integration and system testing, and finally operation and maintenance. A failure in one category causes the next step to go backwards and start the cycle at a different point to fix found problems. The drawback to this is that change is hard to accommodate as the processes aren't supposed to mix. Each phase must be completed before starting the next. Since the process doesn't have a tangible delivery before the product is delivered at the end of the development process, it may take a while to get paid for the work. Another drawback is that there is a large timeframe between requirements gathering and delivery of the product. Since the client isn't getting incremental delivery it's possible to get to the end of the process and have a mismatch between requirements and what the client needs. This causes rework and is expensive to recover from.

   In contrast, incremental development interleaves processes. Starting with an outline: description, specification, development and validation are all done concurrently producing initial and intermediate versions followed by a final version. Unlike waterfall this method allows quicker and cheaper response to changing customer requirements due to incremental delivery. It's easier to get customer feedback on development work as prototypes are produced. This also produces more rapid delivery and deployment of software to the customer and as a result the customer can gain value from the software sooner using functional prototypes. A result of incremental development is the possibility of being paid sooner than if the waterfall development process is used. The drawback to this process is that the relatively small amount of documentation as compared to other processes makes the development less visible. Since it starts with an outline and not a nuts and bolts plan it's harder to show the client that work is getting done before a prototype is created. It's also not possible to tell the client what will be delivered in each increment.

Process-wise incremental development would be preferrable because in my experience at least clients don’t always know what they want and pleasing them can be an expensive endeavor when they change their mind. Being paid sooner is also very important if you rely on contracts for income as a team of people can cause a lot of overhead costs and dragging production timelines due to rework can decrease productivity. This in turn can increase costs even more. The programmer in me however prefers parts of the waterfall method. When I systematically plan the architecture of my programs, they are generally much more useful and less error prone. With the advantage of automatic unit tests however I’m confident this might no longer be the case. Having precise specifications in most industries also seems unrealistic, which makes incremental development preferrable to waterfall at least when working with a client on non critical systems.

8. The requirements phase is one of the most important of the software development phases because it lays the foundation for the rest of the project. Even in incremental development (or its derivative evolutions) having incorrect requirements can cause a lot of rework. If the problems aren’t found early, they can become expensive quickly. Projects can spiral out of control and take twice as long to complete had requirements gathering been properly prioritized. This can strain relationships with the client as well as internal relationships with the team.

   My teams strategy with the requirements document was using a shared google doc and working concurrently. Each team member completed a section of the document that we each volunteered for. As I have less of a heavy workload this term, I dove in quickly to do functional requirements. During this process we all peer reviewed each others work in waves and communicated in the doc via comments about specific sections, and over discord about broader scheduling and deadlines. This involved collaborative discussions on what each section should cover based on what we learned in class as some sections required creative thinking. Doing this via a collaborative shared document was important to us as we have wildly different schedules. We were able to meet during an initial video call, and broadly completed most of the project in two pushes, one on each weekend. The first ended up being an initial draft, and the second push was refining and polishing.

I was able to do a first draft of functional requirements after diving in early, however we needed clarification on some requirements. Before we had clarification everyone worked on their sections concurrently. I worked on functional requirements, Bailee worked on product overview, Jeremiah worked on nonfunctional requirements, Dylan worked on milestones and deliverables, and Jesse completed the introduction. While this was happening, our answers came in via email and we were able to update the document with current information.

The introduction itself didn’t require many revisions and everything was unanimously approved. Jesse did a good job on this and continued to improve it through iterative editing. Our team worked on this section and others making sure that the terminology used was the same as the clients. Initially some of our document referred to toe members as patients.

The product overview was similarly well done, and Bailee built this section out over time eventually adding a diagram after getting group input on what should be included in the diagram. She was instrumental in keeping the team moving, and had a thoughtful contribution to every section.

Functional requirements was a by the book adaptation of the assignment sheet, or so I thought when I completed my first draft. It went through several revisions as requirements were clarified. I found I had misinterpreted the requirements a few times, and was willing to entertain that I could be repeatedly wrong. Having an open mind in a team environment is incredibly important, and my team members really pulled through with insightful observations that allowed us to course correct as a group. I was happy to be wrong as it decreased our workload in future stages of development. My general strategy was to break the work down by each type of user, and add a new section where requirements overlapped.

Nonfunctional requirements was a creative problem solving exercise and Jeremiah stepped up to the plate. In addition to sections covering the obvious non-functional requirements of the software, he also added nonfunctional requirements for the development process, where the data is stored, regulatory and legislative requirements, and ethical requirements. Contribution wise we all reviewed this for accuracy given we were still gathering requirements, but Jeremiah took care of most of it. He frequently came through with observations about the project that corrected my errors and continued to improve his section throughout the project. Clarifications about the project overall helped to shape this section as well.

Milestones and deliverables was implemented in stages by Dylan. Starting with an initial outline, suggestions were made by team members and other sections were added. It was important to us that this section have a little specificity while allowing us flexibility later in the design and implementation processes. Dylan pulled through completing his section and did some solid work.

Sources Cited

Sommerville, Ian. *Software Engineering*. Boston, Mass. Amsterdam Cape Town Pearson Education Limited, 2016.

Amazon Web Services. “What is continuous integration?”

https://aws.amazon.com/devops/continuous-integration/.