Final Exam – CS 410, F20 Email to siyengar@winona.edu Due:Midnight, Wed 12/9/20

What did we do in this class? How do you know that? (5)

We studied introductory topics of Software Engineering. Time was our biggest limiting factor as it we couldn't employ what we learned in lecture and from the textbook in 'realistically' long projects, but were limited to shorter projects. That isn't to say that what we did learn wasn't good enough; it's just that more time would've been quite beneficial.

The projects that we did work on, and the resulting documentation, proved our own learning, and more importantly understanding. Everyone that spoke on the subject seemed to agree that Software Engineering is a worthy endevour, and that proper design and documentation before programming is not only extremely valuable, but necessary in today's commercial software environment. To me, this is far more important than the technical skills of software engineering and design that we accrued over the semester.

What was your significant contribution during Sprint II? (5)

Much of the design work I put in during sprint I carried over, but I was also the one to modify the Diagrams that we had originally created as a group. As far as implementation during sprint II, I focused on the following:

1) Add automatic blank pages to any signature that does not have enough input pages to fill itself

2) Calculate how many signatures a given input PDF requires based on the defined layout, and loop through that signature creation accordingly

3) Assisted Henry in cleaning up temporary files after imposition is complete

4) Stephanie and I mostly split the creation of the slideshow presented during our sprint II demo, with a little help from Henry.

5) Together the three of us finished up the Agile and Scrum Sheet, as well as the README document prior to turning in, with a little help from Akin.

Specifically – what technical and non-technical skills did you gain through this class? (5)

Technical: UML, Git, Design with groupwork in mind

Non-technical: Presentation, design communication, group organization (semi-technical too)

Most of the projects that I've worked on I do alone, and I design them as I go. I certainly think it through and have a good idea of what it will look like, especially regard what the objects are, but this class gave me experience in written documentation and design prior to even starting the coding. It also surprised me a little bit how different that can be in a group. I was an unofficial lead for the group because of my greater experience in coding, and in the the topic of our project, which gave me a rather unique viewpoint and experience in project work. At times it was frustrating and I wanted to work alone, but in general I was glad for the help. I can't say I was surprised by that, but it was certainly valuable to actually experience.

State in one sentence – what is the most significant responsibility of a software engineer. (5)

A Software Engineer must interpret the software needs of a client/userbase into the designs of an adequate software system.

- Designing a system is certainly important, and very time consuming, but I think the most significant part of that is being able to understand and communicate with the client or intended userbase to so that the needs of the system are actually understood. The ability to communicate, understand, and preempt the perceived needs of the user is what separates an average software engineer from a good one. This is also the likely to be the most challenging aspect of automating software engineering.

Describe each of the SOLID for principles software design – with examples. (25)

S – Single Responsibility Principle

A class/method should have only one job.

For example, a calculator: calculator.add(x, y) should not print out the answer. It should do the calculation and let another aspect of the program worry about output.

O – Open-Closed Principle

Entities should open for extension but closed to modification.

For example, in the last OS simulation, I had two semaphores each handled differently. I almost created two objects, both semaphores, so as to deal with the difference in reaction. That would've violated this principle. Instead I had two semaphore each reacted to differently by moving that separation into the system itself. It would've been even cleaner to use the Semaphore to define two extension classes for my different semaphores.

L – Liskov Substitution Principle

A parent class should be able to be substituted for any child class.

For example, a child semaphore class should not add any additional public methods, but only modify those that already exist. Those existing methods can either be added to or completely overwritten while maintaining this principle.

I – Interface Segregation Principle

An interface, and its accompanying attributes, should not implemented unless all parts of it are used.

For example, a Shape class might include a calculateArea() function as well as a calculateVolume() function, but since a 2D shape can only use one of these, that interface should be split into two separate classes like Shape2D and Shape3D.

D – Dependency Inversion Principle

Higher Level Modules should not rely on Lower Level Modules.

In many ways this is an encouragement for abstraction. For example, in Parvata Imposer, there is an OutputPage object and InputPage object, both of which extend a Page interface. Both objects refer to other Input or Output pages, which means that they violate this principle. If OutputPage or InputPage

was changed, the other may also have to be updated. Instead they should use other Pages to refer to those children.

Read up on DevOps: Then write a 1-page essay on the reasons, history, and evolution of DevOps, and then list the Advantages and challenges of DevOps. (30)

It's easy to segregate, and even overlook, the supporting operations of a software project. What kind of system will this software run on? Where is that system located? What other software is running on that system? Is the internet there good; the power reliable? Is that system anywhere near those that (will) use the software you're developing? Each of these, and many other, questions factor into the end result that users will interact with, and thus affect the measurable success of a software project. In short, DevOps is a blending of existing best practices for software development and the relevant operations, throughout the entire life cycle of a given software project.

The Enterprise System Management movement of the mid 2000's pushed those working in operations to try better. Operations as a discipline had been around for some time, but it grew so quickly that the methods involved didn't grow with it, and larger organizations largely had their own specific methods. Around this time, there was also significant growth in smaller organizations calling for better operation methods, as operations became cheaper and web technology more mainstream. In the later 2000's there were even conferences based around operations, in which there was a major push to find methods more flexible than what was currently common. Web environments in particular were not able to adapt as quickly as operations required. At the same time, some notable companies (including Flickr) made internal pushes for collaboration between development and operations, which aided in achieving this flexibility.

This was also the time when Agile practices became common. Many of the newer methodologies here and in operations stemmed from those that saw success in manufacturing settings. So, as development and operations both grew a lot separately, combined with a push for the two fields to work together, we got DevOps. Patrick Debois and Andrew Shafer got together and started talking up this combination of agile methods across both aspects of a project and named it DevOps. Society changed very quickly in the past few decades as digital technology took over nearly every aspect of our lives. To many, DevOps was a relatively natural response to the inflexibility, and resulting slow pace, of previous methodologies.

As a result, software projects are generally treated as living projects. From a user's perspective this often key to a successful project. This allows a software to remain active and relevant, resistant to external change. Software engineers produce better products because they have an understanding of future deployment, and vice versa for operations managers. Project deployments tend to go smoother with less trouble, because errors have been preempted, and the supporting group is more reflexive in response to unforeseen problems.

However, it's not without its difficulties. Any shift to new development practices is going to cost quite a bit, and a change this big goes far beyond updating development documents. Switching to DevOps is an entire cultural change, that would require new training, documentation, tools, and strategies. Once it is in place, this may limit qualifying applicants for new positions, and there has been fear of Operations jobs being taken over entirely by software engineers. It also has the tendency to increase the time cost of initial project development, making it especially difficult to establish for smaller startup companies that don't have the resources to invest, yet.

Read up on CI (Continuous Integration) and CD (Continuous Deployment). Then write a 1-page essay describing the reasons for CI/CD, and the primary tools and techniques used in this effort? (25)

"Long ago", deployment time was even more hectic than it is now. All of the various pieces of code from each person/team had to be painstakingly merged together into a final product. That build then had to be exhaustively tested, bugs fixed, and the whole cycle repeats until the product is satisfactory. It's hard to estimate how long that will take, and many of the issues that do arise could've been fixed long beforehand. Enter Continuous Integration.

Continuous Integration involves a regularly scheduled merging of code, building that code, and running tests on the result. Of course, this isn't as easy just saying it should be done. Code merges require addition effort, time, and enforcement to happen. This also might hinder the development process as 'checkpoints' need to be made more carefully so as to not break anything (It's easier to walk away from buggy code when it's not going to be used until after you fix those bugs. Regular merging means you have to fix those bugs before you walk away). Also, integration, building, and testing often requires an accessible server to host whatever software system is being used to accomplish these tasks, and somebody has to manage that server, which further increases cost, and developer dependency on that system/infrastructure. Part of that system also includes the tests themselves, which need to be written by someone familiar with the software being developed.

As a result, the final product will be less buggy, especially at release thanks to finding and fixing bugs throughout development. Deployment is much easier, and faster, because so much of the work (especially merging) has already been done previously, and since so many bugs have been fixed before launch, developers are free to work on what bug remain much quicker than before. Testing also becomes much more efficient because the vast majority of is now done simultaneously with the rest of development, decreasing the time and monetary cost. All put together Continuous Integration can make for a much smoother and effective deployment of a software system, for a relatively minor investment cost.

To extend continuous integration even further, one can deliver the regularly merged, built, and tested code straight to the consumer. This requires a strong supporting system of continuous integration, including large amounts of automation. It also calls for flagging within the code to disable/enable features as they become usable, to maintain a runnable and at least semi-stable release. Testing also needs to be as exhaustive as possible to avoid common broken releases, which can bog down the process to the point of negating any benefits. Documentation can also slow the process down too much, and as such needs to be well defined and adhered to for the sake of the users.

Certainly a trickier process, but Continuous Deployment can bring even more gains. Deployments are no longer multiday affairs, that stress out and overwork employees and push many systems well beyond their standard use case. More frequent releases greatly speeds up the feedback loop between developers and customers, resulting in a generally better product that fits the immediate needs of the end user. The feedback loop also allows for smaller design decisions to be tested out in the actual user environment, making latestage design decisions less risky and easier to do/undo. These changes, and releases, are easier to fix in broad strokes because a new release is expected anyway, and the broken release likely didn't go out to every user. The users also get to see the improvement as it happens, rather than waiting months or even years to see, resulting in more satisfied users. Like continuous integration, continuous deployment can have some significant improvements on the end product and the development cycle, if the developers are willing and able to implement these ideas.