Barak Barclay

Dr. Kristen Walcott

CS3300-001

14 May 2019

The first part of any Software Development Lifecycle (SDLC) is gathering requirements. With our project, this was pretty much a one-time thing since our customers were open to pretty much anything. Every time we asked them what they wanted, they replied with something along the lines of, “do what you want to do.” On the job, this will be different, though. Customers will need and want things, and it’s important to figure out not only what those needs and wants are but to also distinguish between the two. If a customer needs something, it’s important and should be a priority. If they want something, it can be a stretch goal. It’s also wise to figure out how they prioritize their needs and wants. A lot of times, tradeoffs need to be considered when determining the right approach to a problem – one of those tradeoffs is always time. Customers need to be informed of these tradeoffs and might re-consider their priorities once they know. After talking to a customer, we learned about creating specific, measurable, achievable, relevant, and timeboxed user stories which we did in connextra format.

In the three software development processes, gathering requirements is the beginning. However, this is done more so upfront in the Waterfall process, more so throughout in the Spiral process, and once every one to two weeks with the Agile process. Waterfall tends to have the most documentation, is good for huge projects and works well if the hardware doesn’t change. Spiral allows the customers to see progress and is good for smaller projects with far-apart feedback. Agile is what we used for our class project and is good if lots of customer feedback is needed. If it’s likely that the customer might change their mind, or the customer doesn’t completely know what they want, it’s best to set more meetings with the customer to gather/modify the requirements.

I think the next important piece of the SDLC is project management. In a lot of cases, I think project management should come before choosing the right tools for the job. For our project, we used ZenHub which works well for software development management because it works with the online repository GitHub. GitHub allows for multiple people to work on the same software and minimizes problems of team members interfering with one another. GitHub allows a sort of copy of the master code, called a branch that can be worked on separately from the master code and later those changes can be merged into master. It allows programmers to not have to worry so much about breaking things since they can just revert to master or an old branch. Zenhub allows for project organization and planning and can utilize features of GitHub. It allows the user to break the project into issues, prioritize, gather resources into a single place, shows velocity and shows burndown charts. Velocity shows how much can be done by difficulty and burndown charts show how much work is left versus time. All of this helps break up the project for a team to work on it. Paired programming helped us figure out a lot since we were all unfamiliar with the tools we were using. However, I can also see it slowing progress in other applications if used incorrectly.

The next important part of the SDLC is choosing the right tools for the job. Depending on the project, options may be scarce. Some projects might require certain tools, but where there is leeway, or where leeway can be made, it’s wise to consider what tools are available. In our project, we were required to use the language/ low-level architecture, Ruby. This language was chosen because no one in the class was likely to have experience with it, it’s an interpreted, object-oriented, and dynamically typed language. Interpreted languages are nice because they don’t have to re-compile each time a change is made, but compiled languages have faster performance. Object-orientated languages/approaches are nice because they can model real-world concepts as objects and make programs much easier to write and understand; however, they take up more room in memory and require more execution time. Dynamically typed languages allow the interpreter to deduce the type and type conversions which makes development time faster, but it can also provoke runtime failures. We also talked about Ruby metaprogramming which allows methods and classes can be defined at runtime, modify classes, and create code that is DRY. Knowing these concepts will help the decision-making process for choosing languages easier in the future.

The high-level architecture that we were also required to use was the Rails framework with the Model-View-Controller (MVC) architectural pattern. The Rails framework was chosen because it works well with Ruby. MVC allows for separation of the content from presentation and data-processing from the content. In other words, the model of the data, what the user sees, and how the modeled data is changed are separated and only interact with parts they need to interact with. Alternatives to MVC include the Page Controller pattern that handles requests for specific pages, the Front Controller pattern that handles all requests and the Template View pattern that has markup with interpolation at runtime. Knowing about these architectural patterns will help when it comes to choosing a pattern or using pre-existing code with a pattern any of these patterns.

We were given the option to choose our IDE, but we took the route of less upfront time by sticking with the IDE that the instructions were given for with the homework, CodeEnvy. Had we known how many issues were going to be caused by using it, I think we would've taken the time upfront to get set up in a different IDE. I’m not sure if we really discussed the advantages and disadvantages of different IDE’s, but that will be something to look into when given the choice in the field.

We learned about Behavior Driven Development (BDD) which focuses on designing based on the behavior desired rather than implementation. In BDD requirements are given in user stories. We also learned about Test Driven Development (TDD) which focuses on writing tests first and works well with the Agile development process. There are different levels of testing including unit tests which test individual methods or lines of methods, integration tests that test the interaction between functions, acceptance tests which test what the user interacts with, and system tests that test the program as a whole. We were required to use RSpec for unit level testing and Cucumber for acceptance level testing. Cucumber was intuitive and easy to get used to. Rspec was harder because there are a lot of nuances to it including learning how to use mocks, stubs, doubles, and seams. Testing is important to keep code from crashing, make it secure, snuff out the edge cases (happy and sad paths) and see if your code is working before dealing with long compile times or long download times if downloading to hardware. We learned that tests should be short, do one thing, have few arguments, and have a consistent level of abstraction. We learned how to use different metrics that show instances of poor coding and places where tests should be expanded on including SimpleCov and Flog. Using these metrics, we also refactored our code to make it easier to read, less repetitive, and more functional. When refactoring we know that we learned that we have to keep in mind that tests could stop passing and may need to be refactored as well.

When working in the field, we may encounter legacy code – code we didn’t write and is keeping the program running. Some strategies when dealing with legacy code are to start small, don’t judge, talk to your peers, read variable names, be conventional, and when in doubt, test everything, rinse and repeat. I’m sure there are a lot of other concepts that we learned in class that will be useful when getting into the field as well.