APPENDIX A

Software Development Lifecycle

Whether you are working on a one-person project or as part of a complex multitiered project with multiple teams, you should understand the software development lifecycle. Each phase of the lifecycle has a purpose that will help you write better software. The following phases can be applied to both agile and waterfall project management practices (Figure A-1). The waterfall method is the method where each phase is completed before the work on the next phase begins, like a pool of water that fills up and then spills over falling into the next pool. The agile or iterative method is where software is developed partially, evaluated, and then incrementally adjusted until it is sufficient. This is considered agile because at each iteration the project can change direction to better serve the users; in the waterfall method, the project would have to start over from the beginning. The formality of the artifacts and collateral that are produced by each phase will vary by industry and requirements of the projects you may find yourself working on. It is also important to remember that these phases are not strictly linear. You may find that you do some planning, some analysis, and some design before completing any one of those phases. Equally important is to remember that every software project is different and these lifecycle stages are guidelines.

P. D. Crutcher et al., *Essential Computer Science*,
https://doi.org/10.1007/978-1-4842-7107-0

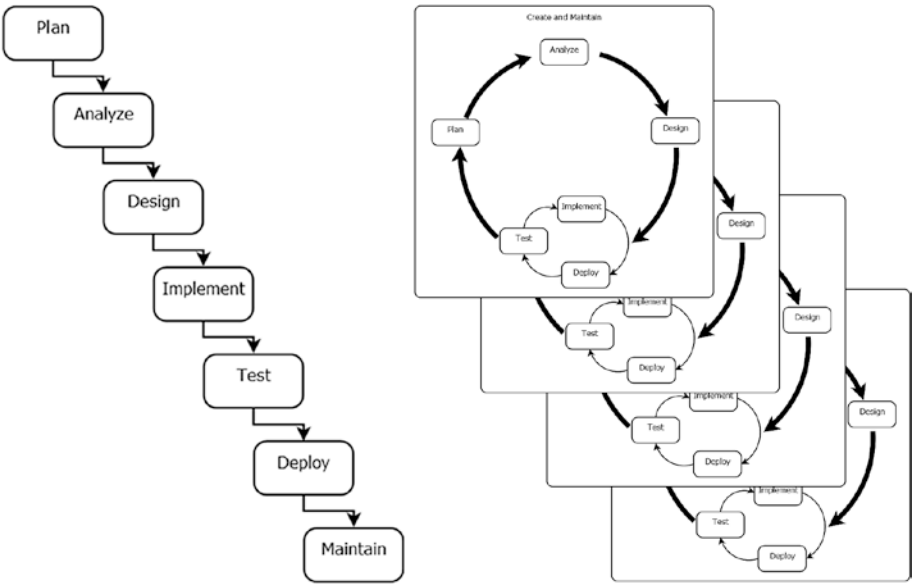

Figure A-1. *Software Development Lifecycle Phases*

Planning

The planning phase is used to determine what software you will create. In planning, you think about what you want the software to do and begin thinking about how you want to do the implementation. Once you have some coherent thoughts on this, start writing those thoughts down. Planning is important for all software development projects, even those using agile methodologies. The detail and length of planning should be determined by the amount of formality needed by the software and/or the industry the software is being targeted to. For instance, planning on an agile project may occur briefly to define what a minimal viable product (MVP) should be for a given iteration. Likewise, on a project for a medical imaging device, much more detailed and rigorous planning may be required.

One important point about planning is that the original plan is rarely what the result will be. It is impossible to predict the future, and plan for every possible change that will affect the plan. There is certainly a point of diminishing returns on planning. Planning will not reveal every possible thing that could occur for your project, nor will everything in the plan materialize. The key to planning is understanding the project and then adjusting to the things that happen in later stages of the lifecycle, without breaking.

Analysis

In the analysis phase, you will define use cases and decompose the problem into logical blocks to help you understand the system, to the best of your ability. Use cases help you focus on how the software will be used; this helps define what the software does and what the users need and prevents creating software that will not be useful. In this phase, you would compare possible algorithms for use in you project, leveraging Big-Oh analysis. This is where you should also develop an understanding of the type and amount of data that your program will be processing. During the analysis phase, you should consider any constraints such as security requirements, usability, cost, feature trade-offs, and long-term support. For instance, if your software will be used over a network, the analysis phase is when you should analyze the network throughput, latency, and frequency requirements for your solution.

The analysis phase may include creating prototypes to better understand the problem. It is important to remember that this is not the implantation phase. Prototypes should be used to understand the problem better and how to approach it. The code that is written as prototypes may not (and probably should not) be included in the implementations.

Architecture and Design

The design of the software is how all the various parts fit together into a consistent whole solution to the problem. Typically we build a solution architecture that lays out the components of a software and how/where they interact. There are at least two interaction areas to cover during the architecture and design phase. First is defining how people will interact with the solution. Second is defining the application programming interfaces (APIs) that define how the components interact with each other. Some software may not have a significant human interaction component, but all software will need to define interfaces (APIs) for access and control.

It's a good idea to do user interface mockups during this phase, to show how a user will interact with the system. If the software is sufficiently complex, various diagrams should be created to help fully understand the design of the software so it can be implemented. A block or object diagram can show how the various components in the software are related to each other. A sequence diagram can show the order that the components communicate with each other and how they interact with each other. A "paper" prototype, mockups, or wireframe diagrams can show what a user might see as they interact with the software. Finally, an API spec should be defined during the design phase to clearly communicate how to interact with the software. The API spec is a key output of the design phase, and it can act as a contract between the components.

Implementation

The implementation phase is where you actually write the software that will address the problems you are trying to solve. You should already have a plan for the implementation and have analyzed the problem to understand the data and algorithms you need. Don't jump into implementation, even on an agile project, without at least some thought

and discussion on the architecture and design. It is, of course, possible to jump straight into implementation, and for the simplest of solutions, that might be ok. But even simple projects will benefit from a lightweight application of planning, analysis, and design.

The technologies and programing languages for your implementation may already be determined for you as constraints of the environment or business. If not, use what you learned in the analysis phase to choose your technology stack.

Test

Testing your software is important to demonstrate that you have indeed solved the problem (verification) and that you have not introduced any new problems or so-called "bugs" (validation).

With the practice of test-driven development (TDD), the test phase and the implementation phase are combined. In TDD, a test is written that will fail until the software is implemented to pass the test. Then the next test and the next part of the implementation are created and so on. More commonly tests are created after the implementation is complete.

Most tests should be written so that they can be run automatically. There is likely some level of testing that cannot be easily automated. These tests should still be documented like a checklist so that the procedure to run these tests can be repeated.

The goals of testing are to discover errors in the software that can have adverse effects on users and data, for instance. Testing can also prove that the software does what is expected. Coverage is a concept in testing that measures how much of the software is covered during testing. Only the simplest of software can have every possible input tested, so coverage helps us discuss how much of the software is tested, which can help build confidence that the software is valid. There are different types of coverage metrics we can measure to indicate how much of the software is covered.

The most common coverage measurement is line coverage. Line coverage measures how many lines or statements of the software are executed during testing. Another common coverage measurement is branch coverage, which measures how many paths through the code are covered.

Test results and coverage measurements provide us with a sense of assurance that the software we develop will work for the users of the software.

Deploy

The deployment phase is when the software is made available for use. There are many types of software deployment. For boxed software deployment, it is preparing the final (compiled) software for inclusion with installer software on a disk. With the growth of the Internet, this mechanism for deployment is not very common anymore. More commonly new deployments are available for download from the Internet, either through an application store or as OS- and language-specific packages. It should also be noted that a lot of software that is written today is never distributed publicly; it's used inside of companies to automate and/or solve specific business problems.

Not only are there many methods to deploy software; software gets deployed in a variety of cadences. Some software is deployed multiple times a day, some once a year, and some only once. Despite the variety of deployment mechanisms and cadences, there are common things you should be aware of when deploying software, such as software licensing, virus scans, and security checks.

The first key to deployment is to understand what audience or audiences you are targeting. This will determine the type of packaging or installer you need to prepare. Once you know your packaging format, consider automating the mechanism of delivering the package of software to your audience. The second key to deployment to consider is a checklist of actions that need to be completed before deployment. These actions

should include items such as making sure whatever license you release your software under matches the license of the ingredient software used in the making of your solution. Of course, you should verify that your tests have run and are successful. The checklist you define will depend on what your audience needs or desires.

Much like testing and test-driven development, continuous integration and continuous deployment (CI/CD) brings deployment into the implementation phase.

Maintenance

Last but certainly not least is the maintenance phase. This is when you change the software to maintain over time. Maintaining software is a much more common activity than creating new software. While maintaining software, you need to be able to identify what parts of the source code need to change (analyze), make changes (design and implementation), test those changes (test), and deploy the new version of the software (deploy). At times, especially when dealing with software that you did not write, this can be difficult. There are some simple actions you can take in other phases to simplify the maintenance phase. In the design phase, you can design the blocks to have very clear, singular purposes. You can also make sure that certain behaviors are only in one block of software. In the implementation phase, you can follow the design as best as possible. Also, during the implementation, comment your code with information about what you are doing and why you are doing it. Consider these comments as a letter to a future maintainer. Having automated tests from the test phase can help prove that any changes during maintenance have not created new issues.

The software development lifecycle for your project will be unique, whether it is closer to the waterfall model, highly iterative, or something in between. This framework of phases should help you manage a broader set of activities, beyond just writing the code.