Software Development Lifecycle Notes

Software Development Lifecycle Intro

The typical software development lifecycle details how a piece of software is built. It covers inception, creation, and maintenance. This model is very high-level, meaning it covers only the large areas of the cycle. Each area can be further expanded to more specifically define the process.

This cycle does not have to be completed in order. There are many different ways to accomplish each task. Depending on the problem at hand, certain solutions will work better than others.

Software Development Lifecycle Steps

Requirements – This section covers how to define the problem at hand. Here we take the idea of the problem, and represent it into more concrete and real world explanations. For example, we might take the idea of a website, and come up with a list of things the website should be able to do.

Design – After detailing the problem, we work on coming up with how we are going to solve that problem. In design we come up with the exact plans on how we can create a product which satisfies our requirements.

Implementation – This is step where we actually begin creating the solution. Here we work on building out all areas set forth in the design step.

Verification - With verification we work on making sure our solution actually solves the problem. We test the product, look for bugs, and adjust any areas which are incorrect.

Maintenance – After verification, we launch the product. From this point on, we begin working on maintaining the product. Bugs will come up, and new features will need to be implemented. This part of the cycle can go on indefinitely.

Lifecycle Importance

Knowing each step in this cycle is crucial for understanding the rest of the course, and software engineering in general. Every project needs to pay attention to each of these areas to be successful in the long-term.

Requirements Notes

Requirements Definition

Requirements define the problem at hand. In this step we are trying to find out the "what". We are NOT trying to find out the "how". This means we want to come up with a set of goals the system should be able to do, but not try to figure out how to accomplish these goals.

For example, let's say we have the problem of needing a form to track financial transactions. In the requirements we might look to define what exactly this form should track. Maybe we need a name, date, amount, and bank account field. From there, it needs to be stored into a database. These would all be requirements. We don't however need to figure out the best way to do any of that. Those questions are for the design phase.

So to reiterate, requirements are just to define the problem. It's the phase where we come up with a common set of goals that we are working towards. The requirements will constantly be looked at through production to make sure the product being built is actually solving the original problem.

Requirements vs Specifications

Within the requirements phase, we also look at something called specifications. Specifications are in essence "technical requirements". They are used to define the problem in terms of the technical constraints.

By constraints, I mean limitations, or implementations that must happen. For example, if a client only uses MySQL databases, then we have to use a MySQL database as well. This isn't a design choice, it's something that must happen, therefore it goes into requirements.

Requirements - A non-technical definition of something that is required from the system.

Specifications - A technical definition of what is required from the system.

With requirements, we are trying to keep it simple. We don't want to use any technical jargon. It should be understood by just about anyone. Something very simple like "*Should record bank account information*" is good.

With specifications, we can be a little more exact, and use some technical jargon. We still want to keep it as simple as possible though. "*Encrypt the bank account information with at-least a AES 256 standard*". Again, we are NOT trying to design it here. We aren't going into the specifics, just saying the technical standard by which we need to meet. We

will come up with how to do this, which encryption to actually use, etc. in the design phase.

Non-Functional vs Functional

There is one final area of organization we can add onto our requirements. This is the idea between functional, and non-functional requirements.

Functional - Functional requirements are requirements which pertain to the function of the system.

Non-Functional - Non-Functional requirements are requirements which cover areas that don't directly effect the function of the program.

Functional requirements are pretty straightforward and easy to create. All we have to ask is "what should the program do?". Any answers to this question are functional requirements.

For example: the system should collect user information, the system should provide a cart system for collecting items, the system should provide a way to rate and review different products. These are all functional requirements. They detail the way the program should function.

Non-functional however are a little bit trickier. They are typically constraints. These constraints might be implemented by the government (safety regulations), the company (quality regulations), or the client. They are things we have to follow, but don't directly pertain to the function of the program.

An example of a non-functional requirement would be that the program must be coded in Java. This requirement might come from the client, so it's easy to maintain by their engineers who only know Java. This doesn't describe the functional of the program at all. It will still do exactly the same thing in any programming language. This makes it a non-functional requirement.

There are different categories of non-functional requirements to make it a little bit easier. These categories are product, organizational, and external.

Product - Must have behaviors of the product (referring to the code, servers, etc). Example: Product must be coded in Java. Product must use Microsoft based servers.

Organizational - Company policies, standards, styles, rules, etc. Example: The project will be developed using scrum. Every function must be documented.

External - External laws, regulations, trends, etc. Product must use SSL due to European law XYZ. Product should use new technique of XYZ to collect data.

Software Architecture Notes

Architecture Introduction

Architecture is the highest level of design within a system. It is the link between idea and reality. It takes our idea for the system, and creates a plan for it. We focus on only the largest areas of the system here. We want to break it down from idea, into concrete areas to build.

In software, bad architecture is something that CAN'T be fixed with good programming. It is a critical step within the development process. Once we decide on an architecture, we have to understand that it **can't be changed.**

This is the same in the real world. Imagine trying to change the foundation of the skyscraper once it's built. Same could be said about a bridge. Imagine trying to switch it from a suspension to a draw bridge after it's built. These changes would be impossible. The only way to implement them would be to destroy the product and start over.

Architecture Overview

Software architecture is all about breaking up larger systems and ideas, into smaller focused systems. Our first step is to take the requirements, and build an initial architecture. We take this broad set of ideas and guidelines, and have to organize it into functioning areas.

Each of these areas are then put through the same process to break them up into smaller and smaller pieces. Eventually we will have a blueprint for the entire system designed.

Good architecture is hard. It takes a lot of resources to develop correctly. However, this upfront cost is almost always recovered from how maintainable the software is. This will reduce the amount of bugs, and the time to fix those bugs.

Good architecture also helps for faster development and better resource utilization. If we break up the project into small pieces, we will understand how to have multiple developers work at the same time on it.

Another benefit of breaking the project up is the idea of buy vs build. If we have for example 4 development teams. Through architecture design we have broke the project up into 5 different large projects. We go through each of these projects and learn that 1 of them has already been solved before and is for sale on the market. We can then allocate

the 4 teams to the other ones, and just purchase the already created software, saving us time and money.

Architecture Patterns

Pipe-and-Filter

The pipe and filter pattern is a good pattern to use to process data through multiple different layers. The key to this pattern is the ability of each step to input, and output the same type of data. So if you send a set of numbers in one side, you will get a set of numbers out the other side.

This key constraint makes it so you can mix and match the logic in any order and still have the program work. These different filters can also be set up across multiple servers.

There is definitely an added complexity with this pattern. Setting it up can be tricky to get correct. Also, if the data is lost at any step, the entire process is broken.

More Information: https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters

Client-Server

The client-server pattern is one that is quite common today. Every single website and most phone apps use this architecture. With this pattern there are two parts to the software, the client, and the server.

Let's take an iPhone app for example. What you download in the app store is what is known as the "client software". This is the version of the app built to talk to the server. It doesn't store any of the server's data locally. It is just setup to make the appropriate server calls when necessary.

The other part of this is of course the "server software". This is the software that is installed onto a server to receive the requests from the client. The server holds and updates the data. It also processes requests, and sends the data to the clients. Servers have to be tuned correctly, as there can be a near unlimited number of clients requesting information.

This is a great pattern for accessing and updating a single repository of information. It's great of keeping track of accounts, and regulating which data is given automatically.

More Information: https://en.wikipedia.org/wiki/Client%E2%80%93server_model

Master-Slave

The master slave pattern consists of two elements, the master, and the slave. The master is in full control of all slaves associated with it. This is good for a multitude of different applications.

One such application is with duplicate backup servers. You don't want these backup servers all acting independently of one another. This will create a bunch of different states of memory. Each server will have a different set of data. Instead, you have a master server, which is the main server of operation.

The master server is the one dealing with all of the normal day to day operations. Then at some point during the day, it sends a signal out to all of the slave servers to tell them to begin their backup operation. The slave servers all start up, copy the data from the master server, and then go back to sleep.

This pattern is also used with "multi-threading". Here we break up an operation into a bunch of small parts. Each of those parts are given a thread and fed through the CPU. If a CPU has multiple cores, it can process multiple threads at the same time.

We typically have a master thread which controls the creation and tracking of all slave threads. The slaves do exactly what the master thread has told them. The master thread keeps reassessing the situation both creating and deleting slave threads. Once the operation is finished, the master thread ceases as well.

More
Information: http://www.openloop.com/softwareEngineering/patterns/designPattern/dPattern_MasterSlave.htm

Layered Pattern

The layered pattern consists of divvying up program into layers of technology. These layers only communicate with adjacent layers. Let's say we have an architecture with 9 layers. In this model, 8 would only be able to communicate with 9 and 7, 4 with 3 and 5, etc.

This simplifies the communication channels, and helps to better distinguish the areas of the program. Overall, this helps to make the program more maintainable. The downside to this, is that there can be of added complexity in some areas. For example, if you need to send a message from layer 1 to layer 9. There will have to be a function in each layer to pass that message along.

More Information: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

Architecture Conclusion

There is no one size fits all plan when it comes to software development. The process must be taken on a case to case basis. Us, as engineers, seek to find the best pattern or set of patterns which solve the problem.

This process is an iterative one. We come up with an idea, get feedback, rework it, and repeat the process many times. After a series of iterations, we have the architecture that will work best for the problem.

Design and Modularity Notes

Design Introduction

Once we have the architecture of the system set up, we can begin working on the design. The design is where we really plan out our system. We can go as detailed as possible in this step.

The main focus here is to break up the project into subsystems, and modules.

Subsystem - Independent system which holds independent value.

Module - Component of a subsystem which cannot function as a standalone.

For example, let's say in our architecture phase we separated the main project into 4 different areas. We would look into each of these 4 areas and come up with the subsystems that would be needed. We would then go into each of those subsystems and see if it can be broken into more subsystems. Eventually we will reach a level where we then come up with some modules that work together to accomplish the task of the subsystem.

Figuring out where to separate the program is one of the most difficult parts of software development. It's very easy to stick with the attempt. This however is just as bad as skipping design entirely, and doing as you program. This step should be iterative. We should slowly refine and reorganize the systems and sub-modules until they make the most sense. Having multiple engineers with multiple viewpoints working on this will help the process immensely.

Information Hiding and Data Encapsulation

Information Hiding - Hiding the complexity of the program inside of a "black box".

Data Encapsulation - Hiding the implementation details from the user, providing only an interface.

Both of these work to hide the implementation details, and protect the integrity of the data. We want to control the flow of data, and provide the user with an easy to use experience.

Imagine on Facebook if you needed to know how to code to submit a new status update. No one would use it. So what we do is we encapsulate that function. We provide the user with an area to enter in a status message. Once the user hits the submit button, all of the code activates to make that status update permanent. All the user needs to do is enter text into the box and click a button.

With these ideas, we can take the most complex of code, and make it accessible to anyone. Doing this at each step in the design process also helps make the code easier to maintain.

With it implemented properly, we don't need to know the entire codebase to make a change. We only need to know the part of the program we are working on. The encapsulation of all the other levels of the program, make things easy to test and change.

More Information: https://techdifferences.com/difference-between-data-hiding-and-encapsulation.html

Coupling Notes

Coupling Introduction

Coupling is one of the major things to look at when designing the modularity of the system. It details how dependent each module is on every other module. A set of modules with tight coupling is bad design. It creates hard to maintain code.

Think about designing a program, and just creating a random set of files to detail that system. There are let's say 1,000 files that make the system work. Each file uses information and calls from 10 other files. Let's say we want to take out a file and replace it with a new one.

At a minimum we will have to update 10 files to make this change. What if one of those calls goes one layer deeper. We have a possibility of 10 * 10, or 100 files that might need to be changed.

This will make the program very inflexible. After it's created, it will only be able to serve one purpose. Any changes will require a ton of effort and money. Overtime the program will quickly become outdated due to this cost.

Our enemy with all of this is dependence. We don't want our modules to be dependent on one another. We want to be able to swap one module out with another, and only have to update code in the swapped module. The more dependent our program is, the more and more modules we will have to rewrite for every change.

Tight Coupling

This is the worst form of coupling. Tight coupling means there is a strong dependence between modules. Changes will be very hard to make and bugs will be difficult to track down.

Content Coupling - This details when one module modifies or relies on the inner workings of another module. If we have module A and module B, tight coupling would be A calling B.data. A in this situation is directly calling the data from B. This means if B were to change, be replaced, or even just be renamed, then A would fail. If a bunch of files all called the information directly from B, and B were renamed, there would be failures in each of these files.

Common Coupling - This is when a bunch of modules have access to manipulate the same global data. If 10 files directly read and write to this global data, we can quickly run into issues. Let's say one of the modules is built wrong. It pushes up an incorrect value to the global data. The other 9 files pull this data down, and instantly corrupt. We now have errors in 10 different places in our program. The only way to track an issue like this down, is to check all 10 files. It becomes a nightmare to debug.

Also, if that variable in the global data is renamed, we have to update all files which touch it as well. Same if we want to split up, or rework the global data. We end up running into a problem where a simple rename project could take weeks.

External Coupling - This coupling is when several modules have direct access to the same external Input/Output. A common example of this is with an API. Let's say we have a program with 800 different files all making API calls to Google.

The issue here is we don't have control over Google. They are free to change their API whenever they want. One day Google does just that. They decide to rename a call we use. We now have to update 800 different files to mitigate this change. More importantly, our active code will be errored out until we can make those changes. If we are running an online store, that can be a very large amount of money lost.

Medium Coupling

Here the coupling is getting better, but we still have room for improvement.

Control Coupling - This is when data is passed that influences the internal logic of another module. A common example of this is through the use of a flag system. Let's say you have a module X which calculates a value depending on the country you are in. So module X must take in the data, and then a country flag to work.

Any module which accesses X will have direct control over it. If we go into module X and change one of the values, we have the possibility of breaking every single direct call.

A better way to do this would be to enter in a latitude and longitude as variables. This way, module X is free to process that data how it wants. It can select the country from within, and make the applicable changes. It can switch out it's code format, and nothing will break.

Data Structure Coupling - This is when multiple modules share the same data-structure. We are starting to get into the area where we might be able to justify this action. Here the major problem is if we switch out the data structure.

If we have multiple modules updating the same Array, there will be a problem if we decide to change it to a linked-list. We will have to update every module. These modules all become dependent on the data-structure. If we find out a new data structure will improve our program, we will have to spend a lot of time in the change.

Loose Coupling

Data Coupling - This is when two modules share the same data. This is a good form of coupling. At the end of the day, our program's modules have to communicate with one another. With this form of coupling, the only thing we are passing back and forth is data. Our modules are processing the data, and making their decisions independently. Due to this, our dependency overall is extremely low.

We have the ability to swap our modules easily, with little code that must be updated. Our program becomes more flexible, and easier to maintain.

Message Coupling - This coupling is when messages or commands are passed between modules. This is when we send a message to another module that tells it to start/stop, or run a function. We aren't controlling how these operations are being implemented. We are just giving the modules messages, which it then interprets and executes through it's own logic.

No Coupling - No communication between modules whatsoever. This is undesirables and unrealistic. Our modules need to communicate to have a purpose. No communication is purely a theory level. Our goal is to get as close to data and message coupling as possible.

Cohesion Notes

Cohesion Introduction

Cohesion is the other area to focus on when we are talking about modularity. Cohesion is the measurement of how focused our module is on a single task. The more focused the module, the higher the cohesion.

With cohesion, higher is better. We want modules which only do one thing and one thing only. The reason for this is with maintainability. Imagine if the entirety of Facebook was on a single file. This would be extremely hard to maintain. There would be millions of lines of code read through to fix a single bug.

Not only this, but we couldn't reuse that code anywhere. If however we have a module which "swaps two values in a database", then we can reuse that in other projects. If we are smart and build a set of really cohesive modules, we can use those modules as building blocks for our next project.

Our goal with modularity is to balance both coupling, and cohesion. A single file is not very cohesive, but it's perfect coupling. 1000 tiny 1 line modules which are linked to other modules are highly cohesive, but very tightly coupled. What we want to find is the point where we can maximize the effects of both. We want to create loose coupling, and high cohesion.

Weak Cohesion

Coincidental Cohesion - The tasks within the module are only linked because they are in the same module. This is the weakest form of cohesion. Here, the modules are completely random. There is nothing linking the tasks within a module, except the fact that they were simply put into the same file.

A single file project would be an example of this. The only reason everything is in the same file, is because no additional files were created. Essentially there is no organization.

Temporal Cohesion - The tasks within the module are only linked because events happen around the same time. Example a module which does:

- Brush Teeth
- Take Shower

- Eat Breakfast

Here the tasks are only linked together because they happen in the morning. Past that, anything goes. An example in the computer world would be "do shutdown activities". In this command we could be doing server calls, shutting off physical machines, sending emails, updating the front end, etc. The list could include really anything. For this reason, it is weakly cohesive.

Logical Cohesion - The tasks within the module are linked due to being in the same general category. With this level, we have begun to organize the modules a bit more. We have made general categories, and broken up the modules into these categories. Example:

- Drive to Dallas
- Fly to Dallas
- Take Train to Dallas
- Take Boat to Dallas

In the above example, the category here is "getting to Dallas". Flying and driving are completely and totally different however. Their implementations would involve entirely different sets of code.

If our module for example was "backup manager", then it still has a lot of wiggle room. In this controller we could be backing up user data, financial data, cache, cookies, and so on. Additionally we could be running commands that might have to do with backing up. Overall, this is still a weak form of cohesion.

Medium Cohesion

Procedural Cohesion - The order of execution passes from one command to the next. Here we have a relationship of time. This is different from temporal because the tasks are both linked, and essential for one another. There is an order by which these must be executed to work properly.

- Spray car with Water
- Fill out form
- Scrub Car and Rinse
- Dry off Car

You can see that drying off the car before we washed it wouldn't make too much sense. However, there isn't a direct link to task focus in this category. "Fill out form" is apart of the business process. It's true that it happens at this point in the procedure, but it's not as focused as we would like.

Communicational Cohesion - When all tasks support the same input and output data. Here we start to organize by an important factor, data. We are now beginning to organize based on computer science related areas, instead of just "logical" areas. Example module:

- Find author of article
- Find date of article
- Find length of article
- Set content of article

Here we are accepting an article as input, and returning a value about that article as output. This module is very specified, dealing with only articles and their content.

Sequential Cohesion - A combination of the previous two. When all tasks work in which the output data for one, is the input data for the next. With this, we have a procedure of tasks, and these tasks all share the same data. Example module:

- Sand car body
- Apply primer to car
- Spray main coat to car
- Spray clear coat to car

With this example, we take a car object and pass it into the first. After this task is complete, it takes the modified car and sends it to second. Then the car goes to the third, and finally to the last. In this process, the tasks are not only linked through a time requirement, they are also linked through a data requirement. The level of organization within the module has increased quite a bit from the start. However, we can still do better.

Strong Cohesion

Functional Cohesion - This is when all tasks within a module support activities needed for one, and only one problem-related task. In essence the module only does a single action. Examples (each are separate modules):

- Determine monthly payment
- Compute intercept of graphs
- Backup user table

In these situations, the module is more like a large function. Just from looking at the name, you know exactly what the module is doing.

Object Cohesion - This can either be lumped in with functional cohesion, or by itself. Object cohesion is when all activities modify a single object.

This only works in object-oriented languages. An example might be a module which only modifies a user object. All tasks within this module update the user module in some way.

Cohesion Conclusion

Cohesion is really important to make code which is easy to understand and maintain. The more focused the modules, the easier the code will be to debug.

Remember however, that this must be a balance with coupling. A group of extremely cohesive modules might also be tightly coupled together. We are looking for the balance between the two. If you remember one thing from these sections, it's that we want "loose coupling, and strong cohesion".

Implementation Notes

Implementation Introduction

Implementation is the step where we actually begin coding our project. We have detailed what objectives the system should meet, have come up with a good design, and are now ready to begin coding.

This process is going to be very specialized on what the system calls for. A website, iPhone app, server OS, desktop app, and eCommerce store will all be entirely different projects. Because of this, in this section I want to focus more on the theory about implementation and deployment. If you want to learn more about the technology that will solve your problem specifically, I might suggest looking up a specific course here on Udemy. There are many great options out there for just about anything.

Implementation Overview

Programmer Care

The number one thing to understand when building software is to take care of your engineers/programmers. When a programmer starts to get tired, their code quality decreases. This decrease in quality can attribute to poor architecture, design, and overall code. This introduces the idea of "technical debt".

Technical debt is a corner cut now, that will cost up to 10x the effort later on. So a 1 hour corner cut now, could cost 10 hours of work down the road. We want to prevent this as much as possible. Imagine if we gained 10 hours of technical debt every week over the course of a 26 week build. That would total to 260 hours of bad work. This means we could be looking at 2600 hours, or 325 WORKDAYS of effort to fix those problems.

We have to remember that 35 hours a week of programming can be just as productive as 70 hours when we factor in technical debt.

Coding Principals

- Use a style guide, so all the code looks generally the same.
- Code is written for people, not computers.
- Make modules easy to learn and understand.
- Go into everything with a plan. (You can experiment, but clean up after)
- Shorter code does not equal better code! MAKE IT READABLE.
- Break up actions into methods.

More Information: https://www.makeuseof.com/tag/basic-programming-principles/

Buy Vs. Build

Good design allows us to see our entire project before it is built. With this, we can decide which areas we want to build, and which areas we want to purchase. The great thing about purchasing code is that it is almost always cheaper.

Imagine a company which develops a subsystem to sell. They spend 3000 man hours building it. At a rate of $40 an hour, they would have spent $12,000 developing the software. If we wanted to build the same thing, we could expect the same outcome in cost.

However, this company wouldn't sell it at $12,000. They have the benefit of being able to sell it over and over and over again. Because of this, they might sell it at say $500. They only need 24 sales to break even, and we save $11,500 through purchasing it.

It's almost always a win-win situation to purchase instead of build. Coding however is usually very specific. This makes it rare to find software that perfectly fits the problem. However, do some research before you begin building, you can save a lot of money.

More Information: https://clevertap.com/blog/build-vs-buy/

Deployment Notes

Deployment Overview

Deployment is the time when we take our program, and deploy it to it's final location. So if it's a website, this is the point we make it public to all. Deployment is an interesting part of the software development process. It comes after testing, but must be planned out, typically before implementation. To get it correct, we must also typically program it a certain way, meaning it also fits into the implementation section.

The level of planning here directly relates to how large the scope of the deployment is. If we are deploying the system for the first time, a lot of planning needs to take place. If we are applying a really small bug patch, then we don't need as much planning.

Deployment should always be designed with the idea of retreat. If something goes wrong, how can we revert. During the initial deployment, this plan might be to take the product back offline. If however we are running a website like Amazon, we will have to do a bit more planning. If we accidentally break the website, we need the fastest plan possible to return the site to it's previous version.

Deployment Rollback

This process of retreat is officially known as "rollback". We look for the point of no return. This is the single point where it takes longer to go back, than it does to just continue with the deployment.

Knowing this point of no return helps us make the decision to rollback, or continue forward. When planning our rollback, we should have key points during the process. At each of these points, we will need to make the decision of whether we rollback, or continue forward.

If a fatal error comes up at any point, even right before completion, we need a plan to get back to the previous version. Maybe this plan entails switching to a backup server, or undoing changes, or having a hybrid version of the two updates running together. Whatever it is, we need to have it planned out beforehand.

Models Notes

Since there is a lot that goes into these models, I thought providing some links to great websites would be better than a wall of text!

Waterfall Model:

- https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm
- https://www.geeksforgeeks.org/software-engineering-classical-waterfall-model/

V-Model:

- https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm
- https://www.geeksforgeeks.org/software-engineering-sdlc-v-model/

Iterative Model:

- https://www.tutorialspoint.com/sdlc/sdlc_iterative_model.htm

Incremental Model:

- https://www.guru99.com/what-is-incremental-model-in-sdlc-advantages-disadvantages.html

Unified Process Model:

- http://agilemodeling.com/essays/agileModelingRUP.htm

Spiral Model:

- https://www.tutorialspoint.com/sdlc/sdlc_spiral_model.htm

Agile Notes

Agile Introduction

Agile is a methodology. It's a way of thinking. It is NOT a model in and of itself.

Agile was invented because the scope of software development was changing. Waterfall methods aren't bad, they're just slow, and not built for a lot of modern problems.

This is largely due to the complexity of modern systems which make them hard to plan 100%. This means at some point, a change will be required, and production will stop until all the documents are updated.

With agile however, we can move a little bit quicker. We are constantly looking for and adjusting to change.

Agile Manifesto

The agile manifesto is a set of guiding principles for agile development. They are:

Individuals and interactions over process and tools.

Working software over comprehensive documentation.

Customer collaboration over contract negotiation.

Responding to change over following a plan.

With this set of rules, all agile models are created. Note, in here we aren't throwing out processes and tools, and we aren't going without a plan. We are simply just

creating priorities within the development process. We want to make the piece of software that is needed. To do this, we need to keep communication channels open and collaborate with all involved.

Through this manifesto, models were created that fit these rules.

Agile Models

Scrum - Scrum is focused on sprints. Sprints are these 1-4 week production cycles. We take the software, come up with a goal of where we want it to be, and then build it to there.

Once we finish a sprint, we then go back to the stakeholders, show them the software, take suggestions, and move on to the next sprint.

All of this allows us to stay flexible. We are communicating with the stakeholders almost every 2 weeks with this model. This means we are constantly able to take those suggestions and change the direction of development.

Just because scrum is Agile doesn't mean it's not structured. With scrum we typically have many assigned roles.

Product Owner **-** In charge of what needs to be done in what manner. Takes inputs from all the stakeholders and helps come up with a prioritization list of what needs to be accomplished next.

Scrum Master **-** Holds team liable to scrum ideas. Facilitates meetings and conflict resolution.

Team - The team building the software. Might be composed of engineers, designers, architects, etc.

More Info: https://zenkit.com/en/blog/scrum-101-an-introduction-to-scrum-project-management/

Kanban - The kanban system is one of optimization. With kanban, we are trying to analyze the flow of production and figure out the slowdowns.

To do this, we usually use some sort of visual flowchart. We break the project up into tasks and fill up the chart. We are then able to see if any part of production has a slowdown. Maybe for example, our review process is slowing us down, or maybe it's planning.

With kanban, we are trying to make small adjustments into the right direction. We want to work with the existing process, not replace it.

More info: https://leankit.com/learn/kanban/kanban-101-learn-the-basics/

Lean Startup - Lean startup is a way of testing out the market before spending on development. Here we create a MVP (Minimum Viable Product) to see if there is interest in the product we are developing.

Production costs a lot of money. It would be really bad if we spent $500,000 on a project, just to figure out that nobody is actually interested in that product.

An example of this would be to build a website that sells a certain product. Get it working to the point where people can place that item into a cart. Then when they go to purchase, have it give them a friendly message stating that this feature will be coming soon. We then track how many people are actually interested in buying products off our website.

If we have a lot of interest, then we are good to go ahead with production. If we don't have as much interest, then maybe we need to rethink our design.

More Info: http://theleanstartup.com/principles

SCRUM Overview Notes

SCRUM Overview

There are many different parts to SCRUM. To operate a good SCRUM framework, each part needs to be incorporated successfully.

3 SCRUM Roles:

1. Product Owner
2. Scrum Master
3. Development Team

5 SCRUM Events:

1. Sprint Planning
2. Daily Standup
3. The Sprint
4. Sprint Review

5. Sprint Retrospective

3 SCRUM Artifacts

1. Sprint Backlog
2. Product Increment
3. Product Backlog

5 SCRUM Values

1. Focus - Focus on the work of the sprint and the goals of the team.
2. Openness - Open and Transparent about all work and challenges.
3. Respect - Respect each other to be capable at their jobs and responsibilities.
4. Commitment - Personally commit to achieving the goals of the SCRUM Team.
5. Courage - Courage to do the right thing and work on tough problems.

3 SCRUM Pillars

1. Transparency - Giving visibility of the entire process to those responsible for the outcome.
2. Inspection - Timely checks and inspections of the progress toward a sprint goal used to detect undesirable changes.
3. Adaption - Adjusting a process as soon as possible to minimize further deviation or issues.

SCRUM Roles Notes

Product Owner

The product owner is the expert on the product. He/she knows how the product operates, the direction it's stakeholders want it to go, and has a roadmap for getting there. Constant communications at all levels is important for this job.

Responsibilities:

- Managing a healthy product backlog.
- Answers product related questions.
- Communicates with stakeholders and the Dev Team consistently.
- Manages budget and release dates.
- Ensures the value of the team.
- Provides feedback at various levels of development.
- Respects commitments.

Scrum Master

The Scrum Master is there to keep the team going and as productive as possible. They are there to be the multiplier. If you have a team which can produce 100 units of work. The Scrum Master is there to try to make that x2 or 200 units. They will do everything in their power to remove blocks, increase morale, and make sure everyone is upholding the tenets of SCRUM to keep things productive.

Responsibilities:

- Facilitates daily standup.
- Removes blocks for the team.
- Keeps the team happy, with high morale.
- Ensures all SCRUM values are being upheld.
- Sets up all SCRUM Meetings.
- Encourages collaboration between everyone.
- Is the mediator, coach, mentor, and guide.

Development Team

The development team is there to create the product. They have the skills and expertise to make whatever needs to happen happen. These members should be as cross-functional as possible. This means everyone ideally should be able to accomplish every task. In programming this means full-stack engineers are best for this role.

Responsibilities:

- Collaborate with Product Owner to create and refine user stories.
- Writing code, and tests to fit expectations.
- Conducting research, design, and prototyping.
- Helping make decisions based on architectures, design, etc.
- Helping to develop and maintain current iterations.