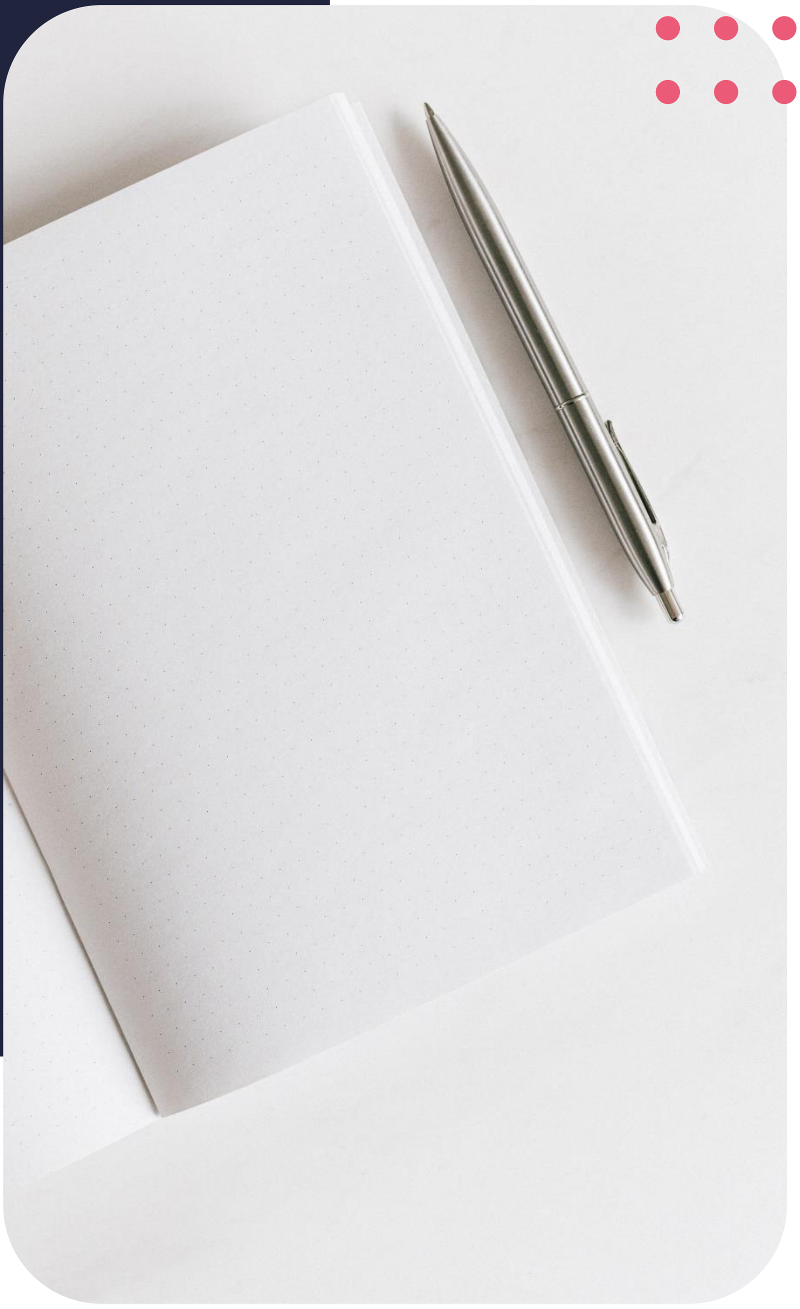
Diploma in Computer Science

The software development lifecycle

Contents



Lesson objectives

By the end of this lesson, you should be able to:

- Appreciate the various strategies employed in software development
- Apply the software development life cycle in writing code
- Explain the various testing strategies that are used when testing code
- Understand the pitfalls that can be encountered in software development and how to avoid themAdjust the width and length of this pink bubble accordingly

Introduction

The software development process

The software development process is the backbone of computer science, system engineering, information systems and software engineering. You will hear this term quite a lot whenever there is a discussion on software development. This term is not only limited to software but applies to hardware as well. You quickly realise that if you're developing a large complex software, it needs to be planned quite extensively in order to make sure that the project is a success. Just like with most other things in science, engineering and even nature, the software development process follows a life cycle that is similar to all of these scenarios.

What are we talking about?

The term life cycle is unashamedly borrowed from nature. Just like the chicken at your favourite fast-food outlet, its journey into existence starts as an egg. The egg grows according to the rules of nature, and four weeks later, we have an adorable little chick. The chick then grows progressively over the course of a few weeks until it becomes an adult chicken. The chicken is then taken do the processing plant, and portions into pieces that are then taken to the fast-food outlet. The fast-food outlet then puts its spices and works magic on the chicken to produce that yummy dish that you at guilty of yearning for all the time!

That's might have gotten you a little hungry, but let's now look at a more fitting description of your life cycle, that of a plant. Plant starts out as a seed. The seed falls onto the ground and becomes a seedling. The seedling grows into a tall and strong plant that then has Flowers of its own. From then, bees come and pollinate it, taking pollen from one flower to another. The Flowers then produce fruits that have seeds in them, and eventually the Flowers dry out and leave fruits. The fruits carry seeds that fall into the ground and the cycle starts over!

The software development life cycle is built on pretty much the same principle. Software starts out as an idea, and developers and engineers get to work, fleshing it out into a problem statement. From then several processes are followed up until we have a full-fledged common functioning software package. The journey doesn't end there. Users need to be trained on how to use the software package, and bugs need to be squashed from time to time. If you have been paying attention to current trends in the industry, you may have noticed that it is no longer enough to just release a single version of a software package and end there. People now expect software to keep improving over time. That's also a way for you the developer to make money as well. Naturally come at this means that the journey will still continue, as software engineers keep looking at the code repeatedly, adding new features, removing redundant features, and improving the efficiency of the code. This continues until the software is deemed obsolete and is decommissioned.

We have already covered bits pieces of the software development life cycle right from the start. As you shall see as we progress with the lesson, flow charts and pseudo code have their own very important places and the software development life cycle. All of this points to one key aspect of software development, which is so important that if you overlook it, you are likely just setting yourself up for spectacular failure. And it's the Golden word, planning.

Reasons for planning

A huge temptation when writing software is to forgo planning and just get on with the work. After all, planning seems like a waste of time when you really know what you want to do. In practise, however, it turns out that planning is not such a big waste of time at all. In actual fact, jumping onto a project without planning is what turns out to be a big waste of time. Not far into the project, you tend to get tangled in so many things and you won't know your way around anymore. Planning enables you to make strategic decisions at the right time, it helps you organise your priorities correctly.

The more complex a project becomes, the more critical it becomes to plan out your work before you do anything. There are things that cannot be skipped, no matter how knowledgeable and experienced you are. Things such as requirements gathering enable you focus your expertise on the right path. There are several reasons why you need to plan your project. Let's take a look at six of the most common ones.

Defining your goals

Like we said at the beginning of the lesson, a project starts out as an idea. Nothing on paper, no code, no fancy graphic user interfaces and no users. At this point, it is important that you set the goals that you seek to achieve. Defining your goals will help you develop the project from that fuzzy idea just something that is clearly defined and outlined with a realistic plan to it. Note the use of the word realistic. Some projects start out with overzealous plans that turned out to be unrealistic. With your realistic plans, your project is more likely to come out successful in the end.

It's the beginning of the project. It's still just an idea. Do you know everything that you need to produce a working piece of software? The most likely answer here is no, and I can confidently say that this is correct more than 90% of the time! Once you define your goals, you need to work out everything that is required, directly and indirectly, to reach that goal. It can come in the form of a team, equipment, money, expertise, or opinions from the end users. Without planning, you’re most likely never going to get your hands on any of this. It is also typical that these requirements change as you progress with your project. Without a solid plan, it is very easy to go over budget and end up with a failed project.

One of the greatest determining forces in a project is its cost. Think of it this way, would you start a business without knowing how much money you need to run things, and how much money you're going to get at the end of the day? This simply just doesn't work. If you've ever approached a bank for a business loan, you'll know that the first thing that they ask for is the plan that you have for spending that money. Nobody wants to sink their money into a hole where they're never going to get it out. Even if you don't borrow money from anyone, it's really just a wasteful practise to throw money at things that you don't know how they will turn out. Planning will also help you to manage costs without sacrificing results.

After creating a basic plan and breaking out everything that you need, you will have a steppingstone that you can use to create a timeline of the things that you expect to do. When it comes to large software projects, it is rarely ever a one man show. Several people need to work on the project at the same time, and this needs to be coordinated. Without a concrete plan or a timeline, there are no deadlines to meet, and as you are going to see as you progress with software development, a perfect, finished software product exists only in fairy tales! It is important that realistic targets are set, and milestones are observed, and the product is released as soon as is humanly possible.

Impact on software quality

It's no secret that, even in day-to-day ordinary life, anything that is planned will simply come out better than something that is just done out of the blue. It allows you to set the boundaries on your project and know when it is time to stop. It also allows you to be better prepared for eventualities. Sorry to say, but those are quite common during software development. Lots of bugs to squash, changing customer requirements, tight budgets, the list goes on and on. Planning the project keeps you on top of things and allows you to deal with unforeseen things without panicking. Planning also provides you with metrics against which you can test the quality of the product. The end result will be a project that is very close to what you will have planned. It is also worth mentioning here that deviations can and will happen, but they need to be controlled in order to make sure that the project itself does not change into something completely different.

Planning allows you to understand the value of your project. Well, it is not easy to tell from the very start, the amount of value that your project will bring, the milestones and deliverables that you set will provide a benchmark that you can keep checking from time to time. It also allows you to manage your time properly and keep track of things such as deadlines and costs.

A bit of history

In the 75 years' history of programming till date, humankind has developed over 700 programming languages. The interesting thing is that the software development life cycle didn't see such phenomenal development since then. The first conceptual framework for structuring software development into phases came into being around the 1970s, in the form of the waterfall model. The first formal description of the waterfall model was cited in a 1970 article by Winston W. Royce. The waterfall model maintained its dominance for well over 2 decades and is still used today by many software developments teams. It provided a way to progress through software development in a linear manner, with milestones in the development process, which made it easy to understand and implement. The waterfall model has distinct phases, and each phase has its own distinct activities. Around the 1980s, a fork war developed in Germany, known as the V model, and was used for defence projects. It is basically an extension of the waterfall model in which there is a testing phase for each corresponding development stage.

This all started to change when software development shifted towards customer centricity, the flaws of the once really great waterfall model started to show. As with everything that is customer centric, requirements often change from time to time. This did not quite work well with waterfall model, which has rigid steps at each stage. Iterative models started appearing around the 1970s to complement the waterfall model's shortcomings. This is what we know today as agile frameworks, which adapt to changes in the user's requirement or development environment. This came in the form of the spiral model and rapid application development. In the 1990s, other models such as feature driven development also appeared. These are really just a variation of agile development. Still on the customer centric basis, other methods such as extreme programming were developed to improve software quality while also catering for changing customer requirements. Features such as continuous integration on site customer refactoring and simple small releases drove the development of these models. Today, agile systems are in widespread use, and it only makes sense because software development is moving so fast that rigid models just can't keep up and would produce software that is unsatisfactory to the customer.

The SDLC

Let's now look at the main topic of the day. The software development life cycle. As I mentioned in the beginning, Software development also follows a life cycle, just like every other thing but you can think of in the world. The purpose of using a life cycle in developing software ensures that it is of the required quality and produced in a timely manner. Nearly all software in the world is now developed using software development methodologies, which all follow the software development life cycle in one way or another. Although methodologies and strategies may differ, the steps in the software development life cycle are pretty much the same across the board. Let's now look at these steps in detail.

Stages in the SDLC

The first stage in the software development life cycle is known as the analysis stage. The main point of this stage is to find out if at all it is a good idea to embark on the project or not. At this stage in the cycle, requirements are gathered from the user, as well as other vital information such as the type of people who are going to use the software package, the type of input the software is going to receive, the data that is going to be produced as output, as well as the setting in which the software is going to be used. The objectives of the project as well as the user's expectations are fleshed out as much as possible at the stage. Another important activity in this stage is to identify the available resources such as developers, the money that is needed to accomplish the project, the equipment that is needed to develop the software, as well as the time that is available. At this stage, communication with the client and developers is vital. If the project is not a completely new one, this is the time that an analysis of what competitors offer in comparison to what the project is going to offer is done. It wouldn't make sense too quiet ahead and spend so much money and effort when you could just walk into a shop and buy a complete software package that caters for your needs! It cannot be emphasised enough that paraplanning is required in order to save time money and resources in later stages of the software development life cycle. At this stage, it is important to define the scope of the project. Although this is not set in stone, setting the scope of the project right from the beginning will help you better plan for the finances that are required as well as the resources. In case you're wondering, scope refers to the upper and lower bounds off the solution in terms of the problem that it is trying to solve. This is because no single software can solve a set of problems entirely. It will take an infinite amount of time to build such a software Package! Setting the scope ensures that you reasonably cater for the customer's needs, while staying within reasonable limits of costs, resources, and time

At this stage, the customers' needs are explicitly defined as computational problems. You recall from module one that every computer program starts off as a clearly defined problem. Notice the word clearly here because ambiguity will have you running in circles at later stages in the development process. We could spell out the stage in three distinct steps

1. Requirements gathering this is where you gather all relevant information from the user and turn it into computational problems
2. Scope definition: the problems are fleshed out as much as possible so that's a rough idea of the upper and lower bounds of the software are apparent.
3. Planning: planning for quality assurance and the risks involved, as well as potential costs, overheads, and time requirements.

The next key stage in the cycle is feasibility study. As the name suggests, we will be looking at whether it is sensible to continue with the project, given all the information that we will have gathered at this stage. At this stage, a document called the software requirement specification, or in short, the SRS document, is produced. It includes everything that is to be designed during the development process. The feasibility study looks at 5 main aspects. These are:

Feasibility

At this stage, we will be checking if current computer systems can handle the kind of system that we want to develop. It makes no sense to develop a system that supposedly does amazing things, yet current systems cannot handle the sheer amount of processing power that is required, or the hardware components that are required simply do not exist yet. The technical requirements also look at what the client has. If you are looking at a small business, you wouldn't expect them to accept proposal that says they should buy a supercomputer in order to use the software product that you develop!

The next aspect is the economic aspect

This looks at the money side of things. We need to make sure that the project can be completed within budget. The budget refers to what the clients can afford as well as what you the programmer can afford. It makes no economic sense to develop a software package that pleases the user but leaves you with a negative bank balance! This typically goes hand in hand with the project scope.

Next, we look at the legal feasibility of the project. You may be technically and economically be able to finish the project, but is it legal? This comes as no surprise as we have seen quite a lot of companies fighting legal fires that arose from their software. This could partly be blamed in the changing of times but could equally be also a result of not doing legal feasibility checks properly. This is a particularly important step, is it just very easy to violate cyber laws and regulatory frameworks, compliances, and policies. This goes from as low as the client's own policies, right up to national and international laws.

Next, we look at operational feasibility. This sounds a lot like technical feasibility but is a more direct aspect of the client's requirements. It looks at whether we can create the operations and processes that are required by the client. The resulting report tells you whether the correct resources and technologies that are required to complete the project exist. It also analyses the technical skills and capabilities of the people that will be involved in the software development process. It will also look at the degree of difficulty of maintenance and upgrades at tertiary stages.

The last and probably the most obvious schedule feasibility. This is where most computer scientists are guilty of not doing things properly! Schedule feasibility looks at the timelines and deadlines which will be followed in the course of development. This makes sure that each stage of development is given a fair share of the developers’ time, without sacrificing quality and deliverables. This is one of the most crucial components of the feasibility study as it will determine whether the project will be completed in time. It is especially important nowadays, because software is now being developed so rapidly that if your project takes too long, it may already have been overtaken by the time you release it! Think of all those apps that you think are so awesome, but no one uses them because after apps came before them and became way more popular. It's hardly the case if you have a client that you are developing an app for, but it is quite possible that the client will get frustrated and walk away. Ouch!

The feasibility study will give you a pretty good picture of the direction in which the project will go at a high level. You can pretty much tell by just looking at the feasibility study report whether it is sensible to go on with a project or not. Now imagine what would have happened if you had just jumped onto a computer and started writing code. It would be pretty disappointing to produce a software package that is irrelevant to everyone and so is not used. This could be said for quite a lot of apps that you find in your device is App Store that provide similar functionality to some other already popular app but do so in a fairly disappointing way and so are hardly ever downloaded. You are pretty much guaranteed to fail if you do not do a proper feasibility study!

After completing the feasibility study and determining that it is sensible to go on with the project, the next stop would be the design phase. This phase produces 2 main documents, namely the high-level design document and the low-level design document. In this stage, that is where you come up with pseudo code and flow charts that's pretty much laid out your solution in a logical manner.

Let's look at each of them in detail.

The high-level design document contains a brief description of each module that is going to be contained in your solution, along with the proposed name. It also shows diagrams that shows the interface relationships and dependencies between modules. In case you're wondering what's that is, it's just offensive description of a flow chart! If your software package is going to include a database, tables are identified along with via key elements. The functionality of every module in the solution is outlined. The document also shows diagram that outline the architecture of technologies that are going to be employed in the solution.

The low-level design document contains all the functional logic after modules, database tables which are quite reasonably fleshed out. These includes the types of tables as well as their size. The user interface is extensively detailed, highlighting key user interaction points and design elements. Any error messages that will be presented to the user or logged as debugging information are documented. The inputs and outputs that we gathered in the first stage are paired up with the modules that will be handling them and documented, as necessary. These two documents will be the base on which actual code is written and tested. This document also extensively reflects the user’s requirements and scope.

The next stage is where people usually want to rush, writing actual code! Reference to the high-level design documents and low-level design documents, developers set out to write the actual code that will be implemented in the solution. If it is a particularly large solution, modules are split up into units and assigned to teams of developers. Typically, this is the longest phase of the life cycle. Constant references made to preceding documents such as the requirements documents and the design documents, so it's not just straight out of the scope and the limitations that will have been set prior to this stage. The programming language will have been chosen prior to this stage according to the feasibility study.

Naturally, the next phase after coding is testing. The code is deployed into the test environment, which mimics the real environment in which the software will eventually be deployed. Testing is carried out using various strategies such as black box and white box testing to name a few. We will look at these more closely in lesson 5. Black box testing entails testing the system at a high level which mimics watch the user will experience. In this method, the behaviour of the system at an external level is scrutinised. This is where errors that are likely to be noticed by the user will be picked up and rectified.

White box testing looks at the internals of the program. The logic, functions, and relations within the system I extensively tested and if any bugs are found are rectified. It instead of picked up in white box testing will typically not show directly to the user, but as would show up as in accuracies in the program’s execution. This stage aims to remove as many bugs as possible, but as we all know, as the scale goes up, it becomes increasingly difficult to create a completely bug free program. The main quality checks are done at this stage. This is not to say that quality checks are not done on prior stages. It is only that at this stage, there is something tangible to quality-check.

After extensive testing and rectification, the team releases a satisfactorily completed software package. This package is then sent to the client for installation or deployment. The software testing stage is not completely over, as there could be new bugs that could be found on site and not in the testing environment. A pilot test is run to see if the system is running as intended, and if everyone is satisfied, that is the developers as well as the customer, the system is then commissioned.

After commissioning the customer starts using the new system. There are still three activities that developers carry out.

As we mentioned, it is not really possible to create a large software package that is completely bug free. Developers from time-to-time release bug fixes to remove these discovered bugs, developers also carry out their own background testing just see if there aren't any more left-over bugs. Any code that is at risk of being broken by new operating systems is adjusted accordingly Developers also optimise the structure of the program so that it runs efficiently. This is called preventive maintenance

Most modern software development doesn't end with deployment. Developers come up with new features that are then added to the existing software. These new features can come from usage suggestions or from the developers themselves. This is called perfective maintenance.

Developers also add new underlying code without necessarily changing the underlying functionality of the software or adding new features. If there's a more efficient way of solving a particular problem, the developers then integrate that your way into the existing software. Developers could come up for example call my change the user interface so that it matches a current version of the operating system. This is called adaptive maintenance

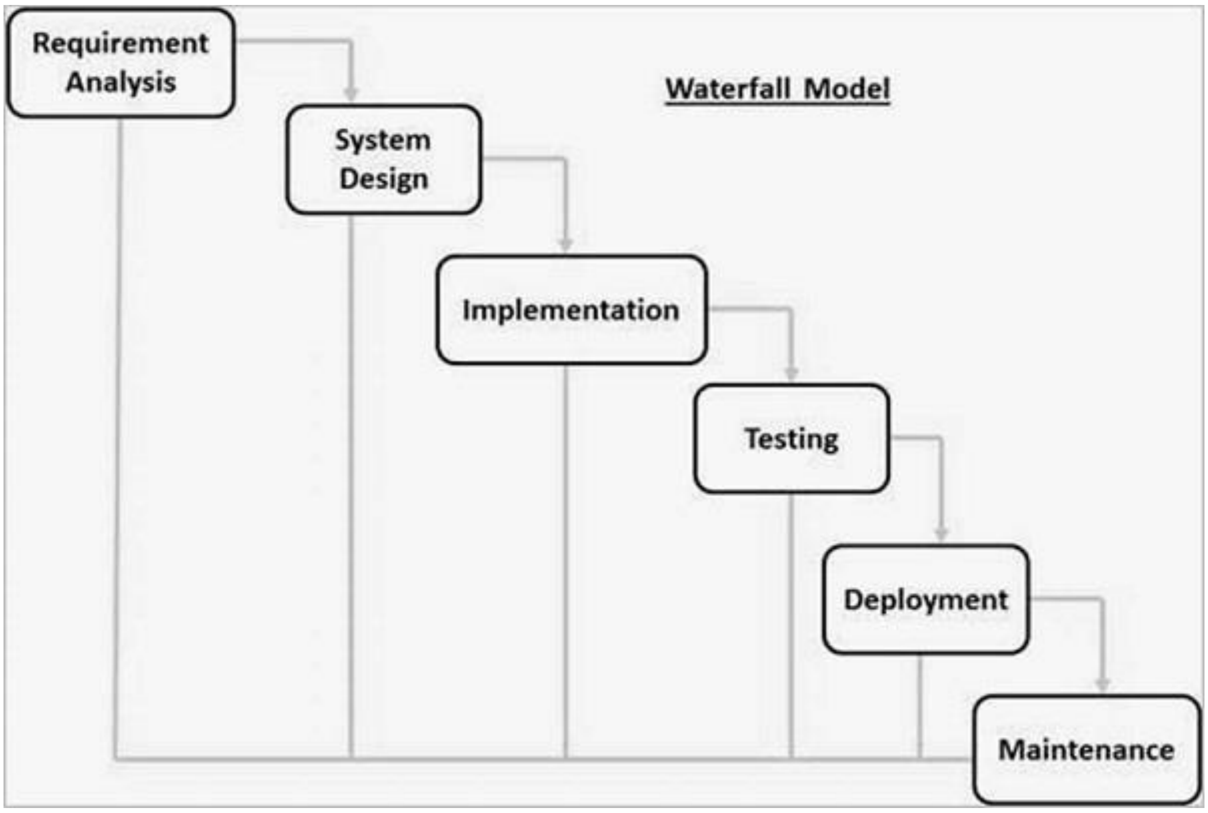
The software package pretty much sits in maintenance for the rest of its life up until it is decommissioned and replaced by something else or just simply becomes obsolete.

SDLC models

There are quite a lot of models that are used in following the software development life cycle. The most popular in probably the oldest is the waterfall model.

Waterfall model

This is the earliest model that was used in software development. It was built on the premise of clearly defining each step of the software development life cycle with clear objectives and well-defined milestones. This diagram shows how the waterfall model executes the software development life cycle.



The waterfall model takes each of these stages as a separate process. A subsequent stage cannot start until the preceding stage hasn't started. Each stage has its own deliverables, that is, a set of tangible results and milestones that are achieved. The waterfall model is arranged in sort of a waterfall, hence the name. It is centred on quite a lot of rigid aspects, namely

- The project is short
- Requirements that are very well documented and fixed
- The definition of the product is stable
- Technology that is used in the implementation and design of the system is stable and not dynamic
- Resources are abundant and the required expertise is at hand

The weather for model has quite a number of advantages as well.

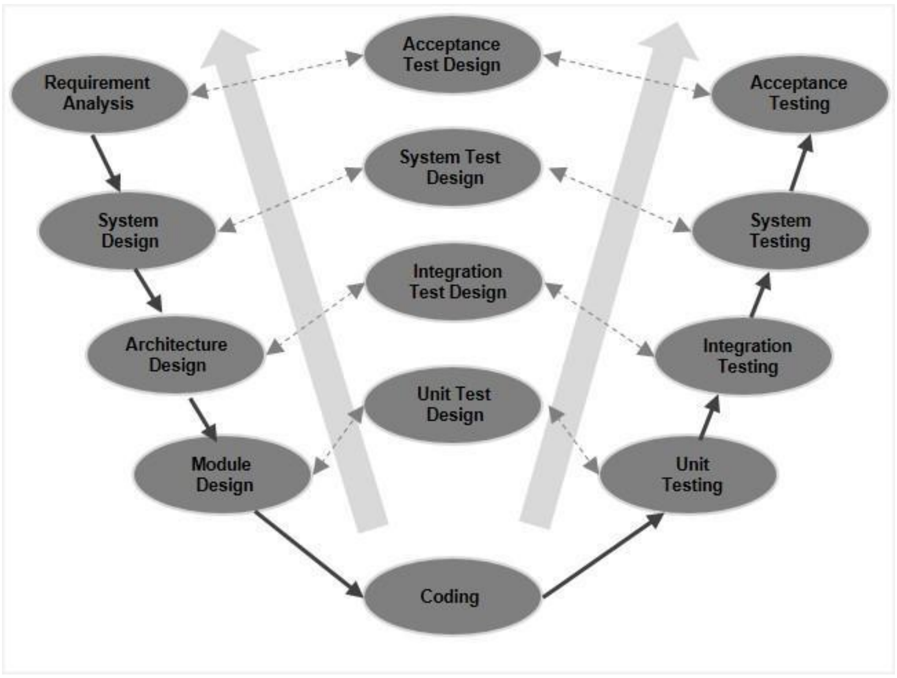
It is a simple model which is easy to understand. Each of the phases are completed one after the other, which makes everything clearly defined. Tasks are easy to arrange and distribute. It also generally produces a well-documented process.

This all sounds awesome, but it is not without its disadvantages.

There isn't any software to talk about until quite late into the production life cycle. This is the direct effect of producing a lot of risk and uncertainty, and problems are often undetected until it's too late. It is not a good model for complex and object-oriented projects and doesn't really work well for long and ongoing projects. If requirements change a lot, it will be difficult to keep track of things. Things such as adjusting the scope are near impossible as you will have to adjust every other component of the project.

V Model

The model is an extension of the waterfall model. It is also known as the verification and validation model. Picture of the steps in the V model is based on the premise that the corresponding testing phase is associated with each of the stages in the software development life cycle. This diagram shows how the V model works.



The model has the actual system development at the centre, and each of the stages correspond to one of the later stages after writing the code.

The V model has the same applications, advantages, and disadvantages as the waterfall model, and to add to that, the testing is carried out in 4 stages.

Unit testing: code is tested at an early stage, module by module, to eliminate bugs

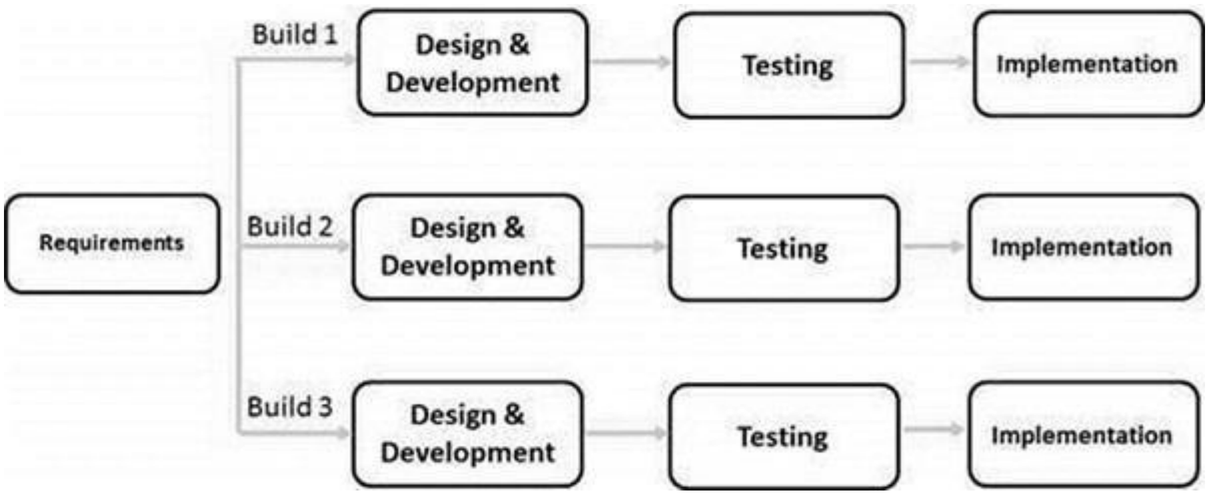
Integration testing: tests the communication of the various components of the program.

System testing: this phase tests the functionality of the entire software package, to check for hardware and software compatibility.

Acceptance testing: here, the software package is tested for compatibility on the host system to see if it will work well with other software that is already on the end user's system.

Iterative development

The iterative model attempts to solve the problems that are encountered in the first 2 models by implementing a small subset of the requirements, then iteratively builds the rest of the package until all the requirements are met.



More than one iteration can be produced at the same time. This quickens the software development process as each iteration builds on the previous.

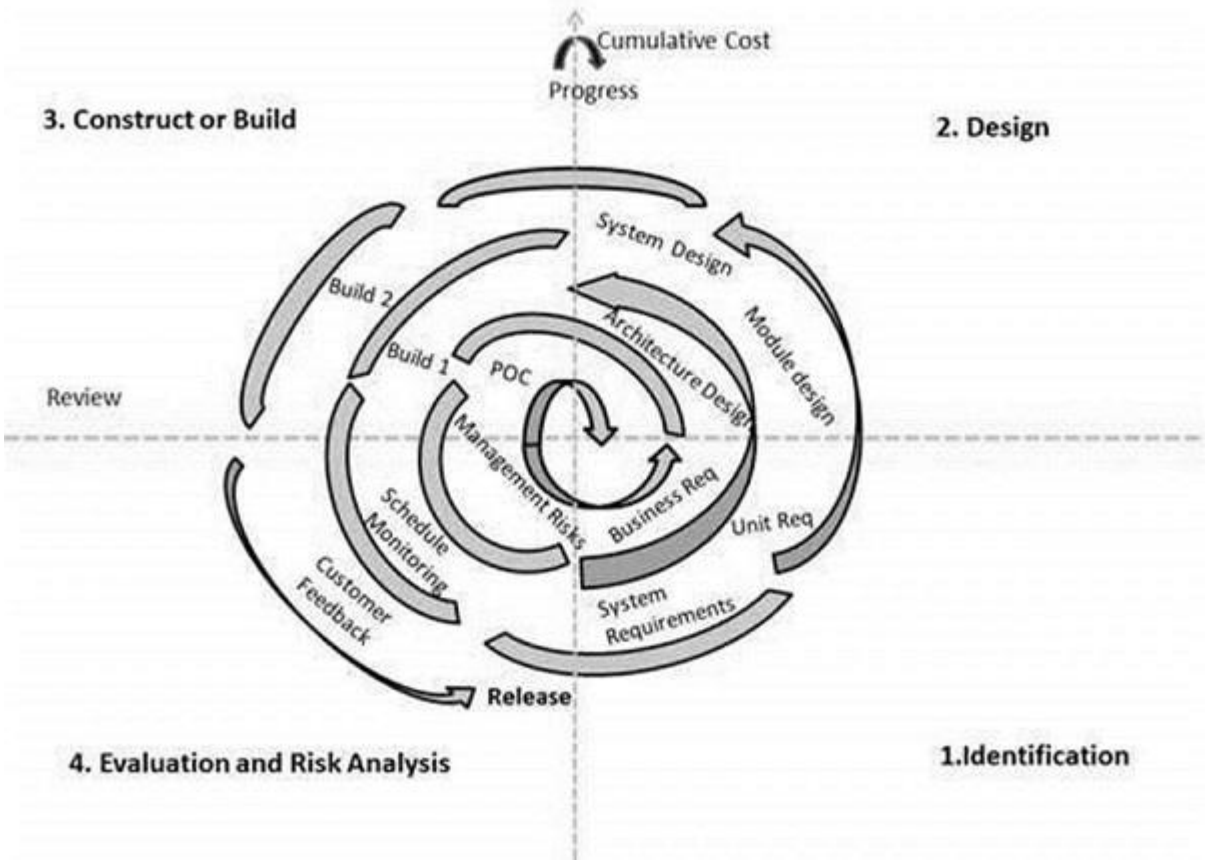
The requirements for the system need to be clearly defined and understood. It does allow for a bit of requirement evolution, but mostly for the minor requirements. It does allow for changes at various stages, including the goals. It has several advantages over the first two models.

It allows for quick development of functionality quite early in the process. It also supports parallel development, and changes to the scope are less costly. The really good thing about the iterative model is that at every stage, there is a functional product.

The model is not without its shortcomings though. Requirements are not as extensively gathered as they are in the two previous models. Also, because there's quite a lot more going on at the same time, management needs to be executed carefully. Also, it's kind of a waste of time when working with small projects. It also requires quite a lot of skill for risk analysis.

Spiral model

Another model that is similar to the iterative model is the spiral model. This diagram illustrates how it works.



The spiral model is quite popular in software development because it mimics the natural development process of a lot of things. It is essentially a mishmash of the waterfall model and the iterative model, with a lot of emphasis on customer feedback. Based on the customer's feedback, each section of the spiral model has its own little waterfall model inside. At the end of the little waterfall, customer feedback is gathered and implemented into the next section of the spiral.

It is a particularly good method to use when they said tight budget and risk evaluation is particularly important. It requires long term commitment because there are a lot of potential changes to the economic priorities Esther requirements change over time. It is also used where significant changes are expected during the development cycle.

The spiral model has the advantage of comfortably accommodating changing requirements. Users get to see what is going on in the development cycle quite early on, and so can provide the relevant feedback. Development can be divided into small chunks which can be better managed.

Things such as time management can become pretty complex in the spiral model, and it is not suitable for small and low risk projects because it works out to be more expensive. Because of the sheer number of iterative processes, it produces quite a lot of documentation which may be difficult to keep track of.

More models and strategies

We have already seen a pattern where nearly all the software development strategies follow the seven basic steps that we discussed at the beginning of the lesson. There are other models that take a completely different approach and deviate from these seven steps. In some models such as the big bang model, there might not even be any planning at all. It is literally just the classic "throw money at your problems" scenario. Even the customer will not really be sure what exactly the final product should do, and things I just picked up along the way. Let's look at some of these methods in greater detail.

Agile development

A common word that pops up whenever you talk about software development is agile development. It is pretty much exactly what it sounds like. Agile development is built upon the iterative development processes requirements and the solution involves through collaboration off cross functioning systems. Agile software development is typically used in development frameworks such yes cramp on my extreme programming and feature driven development. Angel development is based on the principles and values that are aligned in the manifesto for Agile software development, which consist of 12 principles. 12 principles that are defined in the agile development's manifesto are:

1. Customer satisfaction by early and continuous delivery of valuable software.
2. Welcome changing requirements, even in late development.
3. Deliver working software frequently (weeks rather than months)
4. Close, daily cooperation between businesspeople and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the primary measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity—the art of maximizing the amount of work not done—is essential
11. Best architectures, requirements, and designs emerge from self-organizing teams
12. Regularly, the team reflects on how to become more effective, and adjusts accordingly

Earlier models are often referred to as heavyweight programming methods, while agile methods are referred to as lightweight models. Here is a table that summarises this:

Agile Model	**Waterfall Model**
Agile method proposes incremental and iterative approach to software design	Development of the software flows sequentially from start point to end point.
The **agile process** is broken into individual models that designers work on	The design process is not broken into an individual model
The customer has early and frequent opportunities to look at the product and make decision and changes to the project	The customer can only see the product at the end of the project
Agile model is considered unstructured compared to the waterfall model	Waterfall models are more secure because they are so plan oriented
Small projects can be implemented very quickly. For large projects, it is difficult to estimate the development time.	All sorts of project can be estimated and completed.
Error can be fixed in the middle of the project.	Only at the end, the whole product is tested. If the requirement error is found or any changes must be made, the project must start from the beginning
Development process is iterative, and the project is executed in short (2-4) weeks iterations. Planning is very less.	The development process is phased, and the phase is much bigger than iteration. Every phase ends with the detailed description of the next phase.

Documentation attends less priority than software development	Documentation is a top priority and can even use for training staff and upgrade the software with another team
Every iteration has its own testing phase. It allows implementing regression testing every time new functions or logic are released.	Only after the development phase, the testing phase is executed because separate parts are not fully functional.
In agile testing when an iteration end, shippable features of the product is delivered to the customer. New features are usable right after shipment. It is useful when you have good contact with customers.	All features developed are delivered at once after the long implementation phase.
Testers and developers work together	Testers work separately from developers
At the end of every sprint, user acceptance is performed	User acceptance is **performed** at the end of the project.
It requires close communication with developers and together analyse requirements and planning	Developer does not involve in requirement and planning process. Usually, time delays between tests and coding

Agile methods

Agile development consists of a number of processes that are used to speed up software development. These include scrum, crystal methodologies, dynamic software development methods, feature driven development, lean software development and extreme programming. As mentioned earlier, these methodologies are built upon the I iterative development process in order to dramatically shorten software development time, while staffing deliverables at each stage. This is especially important given the pace at which software is evolving nowadays, there is really no time to sit on a project for months or even years on end, whilst expecting to release a fresh, new software package at the end of such a long stretch. Angel development methods are now commonplace, even with large projects like operating systems, as they allow a far shorter turnaround time, while still allowing the development of high-quality software.

The deadly sins of project management

It would be pretty incomplete to talk about software project management and not talk about these seven sins that are the root causes of project failure. Let's look at each of them in detail

Obscurity

When the project plan is not clear, we can't really be sure what we're hoping to achieve, and it is pretty clear that there will be no goals in sight. Goal should always be measurable, specific, actionable, realistic and time based. Communication is of paramount importance in making sure that project goals as clear as possible.

Lethargy

Any lack of urgency in most aspects of life is always a detrimental habit. The project should be planned while the instal time, and clear timelines be set in order to deliver the project in time and make sure that the project is of the desired quality. It can also set in when the people who are sponsoring the project do not really believe in the goals and vision of the project, and this stifles communication.

Insentience

If the project is being led by incompetent people, it often just stumbles from milestone to milestone with no real vision and is most likely set up for certain failure. Projects often lack air communication plan and very little about them is known outside the development team. Search projects are often stillborn.

Tardiness

This is when a project's milestones are not considered with the seriousness they deserve. Deadlines are not met, and the team continues as though nothing has happened. This is very likely to spark frustration on the customer side and search projects might be dropped suddenly. Typically, but sloppiness on milestones comes poor quality, and if the project somehow manages to make it to the end, it will be a far cry from the desired output.

Obsequiousness

This is when the project manager fails to exercise their authority and this leads to problems such as scope creep, budget blowouts and missing milestones. It's just the responsibility of the project manager to make sure that resources are allocated as and when needed and the project is quality checked at every step

Inevitability

The project will have been executed without paying attention to risk. Risk is like a coin. On one side they are positive risks which foster innovation, and negative risks which can spell doom for the project. If both ignored, you can be guaranteed that the end product will be a lukewarm product.

Evolution

This is one of the biggest problems with software projects. Without due care, the scope can creep beyond what team, resources and finances can ever take into account. It must always be top priority to make sure that there is a balance between the quality of the project and the available resources

Conclusion

And with that we have come to the end of our pretty long lesson. You might be surprised that we have covered so much, but this is only a scratch on the surface of what software project management is all about. We could even do an entire course on it! In our next lesson, we will look at how to make our programs communicate directly with the operating system through the command line interface. We will learn how to make the program receiver arguments from the system and also send arguments to the system. That is all for today and I will see you in the next one!

References

Admin (2019). Software Development Life Cycle | SDLC Models , Phases , Methodology. [online] Learn Computer Science. Available at: https://www.learncomputerscienceonline.com/software-development-life-cycle/

Change Factory. (2014). Seven Deadly Sins of Project Management - Change Factory. [online] Available at: https://www.changefactory.com.au/our-thinking/articles/seven-deadly-sins-of-project-management/

Concordia.edu. (2017). 5 System Development Life Cycle Phases. [online] Available at: https://online.concordia.edu/computer-science/system-development-life-cycle-phases/

Cprime. (2019). What is AGILE? - What is SCRUM? - Agile FAQ’s | Cprime. [online] Available at: https://www.cprime.com/resources/what-is-agile-what-is-scrum/

GCSE Computer Science. (2021). SDLC. [online] Available at: http://gcsecompsci.weebly.com/sdlc.html

Linkedin.com. (2018). Evolution of System Development Life Cycle (SDLC). [online] Available at: https://www.linkedin.com/pulse/evolution-system-development-life-cycle-sdlc-shantanu-choudhary

Rungta, K. (2020). SDLC: Phases & Models of Software Development Life Cycle. [online] Guru99.com. Available at: https://www.guru99.com/software-development-life-cycle-tutorial.html