SDLC

The Software Development Lifecycle (SDLC) is a conceptual framework and process model for describing all the activities involved in a process to create, deliver, and maintain high-quality software. The goal of the SDLC is to produce the highest quality software at the lowest cost and in the shortest time possible.

1. Requirements gathering/extraction and analysis. What exactly does the software need to do? How does it need to do it? What other systems will it need to integrate with or share data with? What constraints exist that the solution design will have to contend with?
2. Planning. What tools or software platforms will be needed to build, test, and host the software? What is the project plan? How long will it take? How much will it cost? Who is needed to do the work? How will the software get tested? How will it get deployed?
3. Design. What is the data model for the software application? How will it be architected? What will the User Interface look like? What network and data centre infrastructure will be required? How will the software application solve the business need? How will information security and regulatory compliance requirements be met?
4. Building/Coding. Write the code for the application. Build the interfaces and integrations between the application and other systems. Review and validate code.
5. Testing. Conduct technical tests of the software and infrastructure (unit tests and system tests). Enable end users to conduct user acceptance tests.
6. Deployment. Implement the software in a production environment. Make the software application available to end users for production use. Train end users to operate the software application.
7. Operation. Monitor the use of the software, it's quality of operation (response times, uptimes, error conditions), and its security of operation. Plan improvements as needed.
8. Support/Maintenance. Routine ongoing technical and user support of the software application.

specific implementations of the SDLC in common use, most notably the “waterfall” method and the “agile” method (subjects of a later Task), A recent trend in software development methodologies is the “DevOps” trend, which takes the idea of reducing handoffs between stages even further: the incorporation of operations (“Ops”) and maintenance into the earlier stages of the development (“Dev”) cycle. DevOps seeks to improve usability, relevance, maintainability, and security of the software application through shared responsibility across the lifecycle and the automation of software code versioning, testing, packaging, releasing, configuring, deployment, and operational monitoring.

The waterfall method for software development originated in the 1950s and 1960s as a logical application of existing process models from manufacturing and construction, in an era when software development projects involved large, expensive hardware infrastructure, large and siloed teams of people, and took years to accomplish using early software development tools that made writing, testing, and running code inherently slow and cumbersome.

sequential step-wise implementation of the SDLC was called the “waterfall” method because it looks a bit like a waterfall on a timeline diagram, with each step finishing before the next one can begin, with a formal handoff between them. No development is done until design is finished; no testing is done until development is finished.

Thus, the entire system is delivered to the client all at once, at the very end of a long project, with little to no value delivered before then. The waterfall method reached its zenith in the 1980s.

This method does have some advantages:

- Scope creep is tightly managed or prevented
- Work planning and budget forecasts can be relatively accurate even over a span of years
- Process is straightforward, relatively simple, easily communicated, and doesn’t require training in mindset or process to use it

- Teams that for some reason cannot work closely together or operate with shared responsibility can still collectively accomplish the project with well-documented outputs from each phase as input to the next
- Its very easy to know how far along a project is, who is responsible at each stage, what “done” looks like, and when “done” will occur

But the waterfall method has some significant disadvantages as well, including:

- Small tasks or details unfinished in one phase can hold up the whole process
- Cumbersome and expensive to respond to significant changes in scope or design (which are inevitable over a project lasting months, let alone years)
- Each phase and its corresponding team are relatively disconnected from the others, with the customers (whose needs are ostensibly driving the design and functionality of the software) as far removed as possible from the requirements definition and design phases where their input is most needed
- Large documentation overhead drives up cost and delivery time

Agile method was developed which rethought the whole development cycle to be iterative, incremental, and collaborative with shared responsibility, rather than one-time-through and sequential.

In this model, software is broken down into smaller sets of functionality, each of which is then designed, built, tested, and released in repeating waves or “sprints” in direct collaboration with the customer, who has input into each of those sprints, all along the development cycle.

The Agile implementation of the SDLC follows four key values:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Agile introduced a new concept for software development: the Minimum Viable Product (MVP). The MVP is the version of the product that has just enough working features and functionality to satisfy a customer need and collect maximum information about customer opinion and satisfaction with the least invested effort. By managing your Agile process to build the MVP first, you validate both your understanding of customer needs and the viability of the software with the least cost and the least risk, before iterating the process to add additional functionality.

Advantages of the Agile method include:

- Designed with change in mind. Changes to scope and design are expected and easily accommodated by the process
- Easy and quick to pivot based on changing customer feedback
- Useful value is delivered frequently and repeatedly throughout the lifecycle, starting with delivering the MVP
- Faster to production deployment than with waterfall, with a generally lower overall risk profile and a higher ROI

Agile does have some disadvantages, including:

- Can get too responsive to frequent changes, with too much input from the customer. A good balance must be maintained
- Requires training team members and customers to best implement and utilise the Agile mindset and Agile cultural work processes
- Requires strong communication and teamwork skills to be effective
- Less effective if the customer does not want to be a direct and active member of the team

STLC

Software Testing Lifecycle (STLC) is a conceptual framework and process model for describing all the activities involved in the process to ensure software actually does what it was designed to do, without defects, and support the development team in correcting defects. This assurance is called validation.

QA validation of software occurred only at the end of a development process. As software development itself became more iterative and with more shared responsibility across stages, the stages of the STLC came to mirror and parallel those of the SDLC:

1. Requirements Analysis. What functionality needs to be tested? How can quality of the software be validated?
2. Test Planning. Develop the testing plan for unit, integration, system, and user acceptance tests. Determine test strategy, methods, and resources required.
3. Test Design. Develop test cases, test scripts, and test data. Ensure test cases are traceable back to the requirements.
4. Test Environment Setup. Build and configure the hardware and software environment that will be used to conduct testing.
5. Test Execution. Conduct testing using the test cases, and test scripts if automated testing is being employed. Unit, integration, system, and acceptance tests are performed during the relevant stages of the development lifecycle. Communicate defects back to the development team. Retest once defects have been corrected.
6. Test Closeout. Complete all retests as needed. Analyse defect metrics and test results to document software quality. Prepare and deliver final validation documentation as needed. Communicate insights that can be used to improve the overall development process.

With the testing lifecycle paired to the development lifecycle, more comprehensive and more timely test results are available at each point in the development process, resulting in faster overall deployment of production software with higher overall quality.

Other benefits of this structured lifecycle approach to testing include:

- A systematic quality process that is repeatable, measurable, and continuously improvable

- Defects can be identified earlier in the development cycle
- Security issues can be identified earlier in the development cycle
- Lower maintenance costs
- Higher client/customer satisfaction
- More stable and consistent system performance

To reduce risk and identify defects as early as possible, different levels of software testing are performed at different points in the development cycle:

- Unit tests. A code-level check, performed by members of the development team, to see if individual components of code are working correctly.
- Integration tests. Tests of the connectivity and data transfer between unit tested code components or other integration points in the system. Also often done by the development team, but could also be done by a QA team.
- System tests. Also known as end-to-end testing, system tests are conducted to test the completed, fully integrated software application from a technical point of view, checking that each input leads to the correct output. Often done by a QA testing team at the end of each release cycle. Often referred to as SIT (system integration testing).
- Acceptance tests. Tests performed by a QA team or a team of representative end users from the client to test the usability and functionality from the end user's perspective.   Often referred to as UAT (user acceptance testing). Usually the last step before deployment to production and go-live.
- Regression tests. Systematic retesting of functionality that might have been affected by a modification made to other code to verify no unintended effects of the change. Usually performed by whoever did SIT or UAT testing.

Manual testing uses written test cases that come closest to mimicking real end-user activity and is best used when testing is exploratory, ad-hoc, or primarily focused on usability by humans. Automated testing uses software to execute test scripts to test functionality, speed and stability under load, transport and transformation of data, and any other aspects of the system that can be tested without human intervention.

Advantages of automated testing include:

- Faster to execute testing
- Can perform many test scripts in parallel
- Faster to production deployment and go-live

- Less expensive at scale
- More reliable, reduces chance of human error
- Can run unattended 24/7

Disadvantages of automated testing include:

- More expensive upfront
- Complex technical infrastructure to maintain
- Requires technical/coding skills to configure and operate
- Doesn’t catch usability issues
- Doesn’t catch issues that humans might identify through exploratory testing
- Doesn’t train a team of end users (by doing manual testing) who can then train other end users

Computational thinking

Computational thinking is a structured problem-solving approach that starts with a problem statement and ends with a set of logical steps that can be followed by humans or computers to produce the desired output. Computational thinking involves four key elements:

- Decomposition – breaking the problem down into smaller parts
- Pattern Recognition – analysing data to find connections and relationships
- Abstraction – identifying the key data and relationships that solve the problem
- Algorithmic thinking – designing a reproducible step-by-step process that solves the problem and produces the desired output

Algorithmic thinking is the key skill needed to become an effective programmer.

An algorithm is simply a list or diagram of process steps that, when followed, start with the defined inputs and produce the defined outputs. Algorithmic thinking leads not to a single answer but rather to a process for getting the answer: a series of logical steps that are sequential, replicable, predictable, and reliable. In other words, given the same inputs, anyone using the algorithm will get the same output, every time.

Debugging

concept of code “debugging” goes all the way back to the earliest days of computer science, when computers were room-sized machines that used vacuum tubes and electromechanical relays for switching “1”s and “0”s instead of electronics. In 1947, at the Harvard University Computation Lab, the room-sized Mark I computer’s operation was disrupted when an actual bug, a moth, was discovered stuck inside a mechanical relay, blocking its operation. While the term “bug” to mean a malfunction had been in use for many years in other industries, this incident at the Harvard computer lab led to its use in computer science, and the finding and removing of such bugs became known as debugging.

There are three categories of common bugs or errors in computer code:

1. Logic errors. Errors in the logic of the algorithm itself. Not strictly speaking an error in the code itself so much as an error in the thinking behind the code. Common logic errors include missing or unusual conditions under which a THEN or an ELSE should occur but the code does not include them because the algorithm was incompletely specified.
2. Syntax errors. Errors in use of programming language vocabulary and grammar when writing code. These errors prevent the computer from correctly parsing and understanding the code as written, so the code will not run. Common syntax errors include misspelling function or variable names and leaving out required parentheses or semicolons.
3. Run-time errors. Errors that occur during the execution of the code. Either the code will run but the output is wrong, or the code stops running because a necessary condition doesn’t currently exist, like a file is missing or a database is inaccessible.

Generally these coding errors are found and corrected during unit testing by the development team, but occasionally a code-level error is not found until later system or acceptance testing identifies a problem with functionality, often as a result of an “edge” case, where inputs to the system are unusual or extreme.

How to debug code

There are a wide variety of techniques commonly used to debug code, including:

- Input simplification – input just one variable or bit of data at a time to see which input causes a problem
- Stepping – watch the code execute one line at a time to see which line causes a problem and what the computer was trying to do at the moment the problem occurred
- Backtracking – essentially stepping but stepping backwards from the point of error
- Output statement tracing – add code at various points in your program to print out status or values of variables, and use the printed values to trace where the program went wrong and why
- Divide and conquer – run only one section of code at a time (usually by "commenting out" the other sections so the computer doesn't see them) to isolate the error systematically

Books:

- *Debugging: The 9 Indispensable Rules*, David J. Agans
- *The Joy of Debugging*, Bryan Cantrill and David Pacheco
- *Debug It!,* Paul Butcher
- *Why Programs Fail*, Andreas Zeller