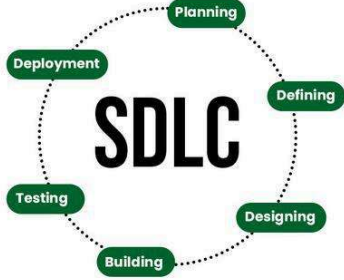


1. **SDLC Overview** - Create a one-page infographic that outlines the SDLC phases (Requirements, Design, Implementation, Testing, and Deployment), highlighting the importance of each phase and how they interconnect.

<p><b>Software Development Life Cycle (SDLC) Overview</b></p> <p>SDLC is a structured process for designing, developing, testing, and deploying software, ensuring a systematic approach to software development.</p> <p><b>1. Requirements Phase</b></p> <ul style="list-style-type: none"><li>- Importance: Define what the software needs to achieve.</li><li>- Activities: Gather, analyze, and document requirements.</li><li>- Interconnects with the design phase.</li></ul> <p><b>2. Design Phase</b></p> <ul style="list-style-type: none"><li>- Importance: Plan how the software will meet requirements.</li><li>- Activities: Create system architecture and detailed design.</li><li>- Interconnects with the implementation phase.</li></ul> <p><b>3. Implementation Phase</b></p> <ul style="list-style-type: none"><li>- Importance: Develop the software according to design.</li><li>- Activities: Write code and conduct unit testing.</li><li>- Interconnects with the testing phase.</li></ul>	<p><b>4. Testing Phase</b></p> <ul style="list-style-type: none"><li>- Importance: Verify software quality and functionality.</li><li>- Activities: Perform various tests (unit, integration, system, UAT).</li><li>- Interconnects with the deployment phase.</li></ul> <p><b>5. Deployment Phase</b></p> <ul style="list-style-type: none"><li>- Importance: Release the software to users.</li><li>- Activities: Plan deployment, rollout, and support.</li><li>- Interconnect with the Requirements phase for updates.</li></ul> <p><b>Key Points:</b></p> <ul style="list-style-type: none"><li>- Each phase builds upon the previous one.</li><li>- Skipping phases risk software quality.</li><li>- The iterative approach allows for revisions.</li></ul> <p><b>Conclusion:</b></p> <p>SDLC ensures systematic software development, from requirements to deployment, ensuring high quality and user satisfaction.</p> 
---	---

## **2. Develop a case study analyzing the implementation of SDLC phases in a real-world engineering project. Evaluate how Requirement Gathering, Design, Implementation, Testing, Deployment, and Maintenance contribute to project outcomes.**

**Case Study:** Implementation of SDLC Phases in a Real-World Engineering Project

Project Overview:

Company: Wipro Engineering Solutions

Project: Smart Building Management System (SBMS)

Duration: 18 months

Team Size: 15 engineers, 3 project managers, 2 QA testers

### **1. Requirements Phase:**

- Objective: Define SBMS functionalities based on client needs.
- Activities: Gathered client requirements through meetings and interviews.
- Outcome: Approved requirements document.

### **2. Design Phase:**

- Objective: Design system architecture and user interface.
- Activities: Developed system architecture and UI wireframes.
- Outcome: Approved design documents.

### **3. Implementation Phase:**

- Objective: Build SBMS according to design.
- Activities: Assigned tasks to teams, used Agile methodology.
- Outcome: Initial SBMS version developed.

### **4. Testing Phase:**

- Objective: Ensure SBMS functionality and reliability.
- Activities: Conducted unit, integration, and system testing.
- Outcome: Bug-free SBMS.

## **5. Deployment Phase:**

- Objective: Release SBMS to client sites.
- Activities: Prepared deployment plans, and provided training.
- Outcome: SBMS successfully deployed.

## **Lessons Learned:**

- Clear communication ensured alignment with client expectations.
- The agile approach allowed for flexibility and adjustments.
- Thorough testing minimized post-deployment issues.
- Training and support facilitated SBMS adoption.

## **Conclusion:**

Following SDLC phases, Wipro Engineering Solutions developed and deployed an efficient SBMS, meeting client requirements and ensuring project success.

## **3. Research and compare SDLC models suitable for engineering projects. Present findings on Waterfall, Agile, Spiral, and V-Model approaches, emphasizing their advantages, disadvantages, and applicability in different engineering contexts.**

### **waterfall Model:**

- Sequential approach with distinct phases (requirements, design, implementation, testing, deployment).
- Well-suited for projects with clearly defined requirements.
- Limited flexibility for changes once a phase is completed.
- Suitable for engineering projects with stable requirements and predictable outcomes, such as construction or hardware manufacturing.

### **Agile Model:**

- Iterative and flexible approach, emphasizing collaboration and customer feedback.
- Sprints: Short development cycles with continuous integration and delivery.
- Well-suited for projects with evolving or unclear requirements.
- Allows for adaptation to changing conditions and customer needs.
- Suitable for software development, research projects, or projects with dynamic requirements.

**Spiral Model:**

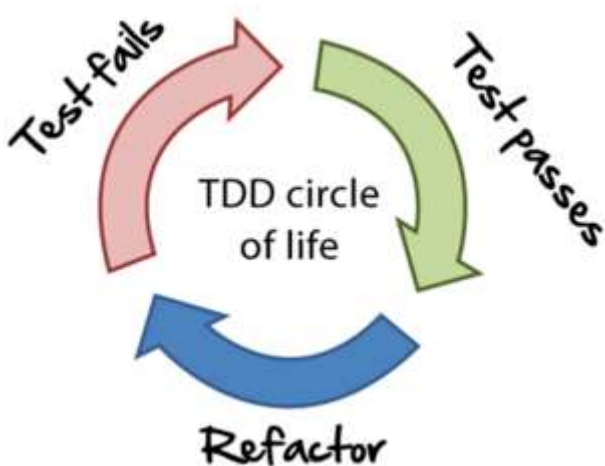
- Combines elements of both Waterfall and Agile approaches.
- Emphasizes risk management through iterative cycles.
- Each cycle includes planning, risk analysis, engineering, and evaluation.
- Suitable for projects with high risks, uncertainty, or evolving requirements.
- Provides flexibility while addressing risks early in the development process.
- Suitable for research and development projects, complex software systems, or projects with evolving requirements.

**V-Model:**

- Extension of the Waterfall model, emphasizing testing throughout the development lifecycle.
- Phases are organized in a V-shaped manner, with verification (left side) and validation (right side).
- Ensures thorough testing and traceability of requirements.
- Suitable for projects with strict quality assurance requirements.
- Provides early detection and resolution of defects.
- Suitable for engineering projects with stringent quality control, such as aerospace or medical device development.

<b>Waterfall Model:</b>	<b>Agile Model:</b>	<b>Spiral Model:</b>	<b>V-Model:</b>
<ul style="list-style-type: none"><li>- <b>Advantages:</b><ul style="list-style-type: none"><li>- Easy to understand.</li><li>- Clear structure.</li></ul></li><li>- <b>Disadvantages:</b><ul style="list-style-type: none"><li>- No flexibility.</li><li>- Late feedback.</li></ul></li><li>- <b>Applicability:</b> Good for projects with clear requirements, like building construction.</li></ul>	<ul style="list-style-type: none"><li>- <b>Advantages:</b><ul style="list-style-type: none"><li>- Flexible.</li><li>- Customer involvement.</li></ul></li><li>- <b>Disadvantages:</b><ul style="list-style-type: none"><li>- Complex.</li><li>- Needs a skilled team.</li></ul></li><li>- <b>Applicability:</b> Ideal for projects with changing requirements, like software development.</li></ul>	<ul style="list-style-type: none"><li>- <b>Advantages:</b><ul style="list-style-type: none"><li>- Good for managing risks.</li><li>- Allows changes.</li></ul></li><li>- <b>Disadvantages:</b><ul style="list-style-type: none"><li>- Time-consuming.</li><li>- Needs careful planning.</li></ul></li><li>- <b>Applicability:</b> Suitable for projects with high risks, like new product development.</li></ul>	<ul style="list-style-type: none"><li>- <b>Advantages:</b><ul style="list-style-type: none"><li>- Clear documentation.</li><li>- Systematic testing.</li></ul></li><li>- <b>Disadvantages:</b><ul style="list-style-type: none"><li>- No flexibility.</li><li>- Complex.</li></ul></li><li>- <b>Applicability:</b> Best for projects with strict quality control, like aerospace engineering.</li></ul>

4. Create an infographic illustrating the Test-Driven Development (TDD) process. Highlight steps like writing tests before code, benefits such as bug reduction, and how it fosters software reliability.

<p>Test-Driven Development (TDD) Process</p> <p>1. Write Test:</p> <ul style="list-style-type: none"><li>- Objective: Define a test for the desired functionality.</li><li>- Action: Write a test that fails because the functionality isn't implemented yet.</li><li>- Example: Write a test to check if a function returns the correct sum of two numbers.</li></ul> <p>2. Run Test:</p> <ul style="list-style-type: none"><li>- Objective: Verify that the test fails as expected.</li><li>- Action: Run the test to ensure it fails due to the missing functionality.</li><li>- Example: Run the test for the sum function, expecting a failure.</li></ul> <p>3. Write Code:</p> <ul style="list-style-type: none"><li>- Objective: Implement the functionality to make the test pass.</li><li>- Action: Write the simplest code that makes the test pass.</li><li>- Example: Write code for the sum function to return the correct sum of two numbers.</li></ul>	<p>4. Run Test Again</p> <ul style="list-style-type: none"><li>- Objective: Verify that the implemented code passes the test.</li><li>- Action: Run the test again to ensure it passes with the implemented code.</li><li>- Example: Run the test for the sum function again, expecting it to pass.</li></ul> <p>5. Refactor Code:</p> <ul style="list-style-type: none"><li>- Objective: Improve the code without changing its functionality.</li><li>- Action: Refactor the code to improve readability, performance, or maintainability.</li><li>- Example: Simplify or optimize the code for the sum function.</li></ul> <p>6. Repeat:</p> <ul style="list-style-type: none"><li>- Objective: Repeat the process for each new functionality.</li><li>- Action: Write a new failing test, implement code, run the test, and refactor if needed.</li><li>- Example: Write tests and code for subtract, multiply, and divide functions.</li></ul> <div></div>
---	--

5. Produce a comparative infographic of TDD, BDD, and FDD methodologies. Illustrate their unique approaches, benefits, and suitability for different software development contexts. Use visuals to enhance understanding.

- TDD: Best suited for ensuring code correctness and reliability.
- BDD: Ideal for projects with complex requirements and stakeholder involvement.
- FDD: Suitable for projects with well-defined features and milestone planning.

Aspect	Test-Driven Development (TDD)	Behavior-Driven Development (BDD)	Feature-Driven Development (FDD)
Approach	Tests written before code	Behavior described before code	Development based on features
Focus	Testing and code correctness	Behavior and system requirements	Feature design and delivery
Language	Tests written in code	Natural language specifications	Features defined in technical terms
Process	Test-first approach	Behavior specification first	Feature breakdown and delivery
Collaboration	Less emphasis on collaboration	Collaboration between teams	Collaboration between developers
Documentation	Tests serve as documentation	Behavior scenarios serve as documentation	Feature lists and design documents
Benefits	Early bug detection, improved code quality	Clear communication, business alignment	Structured development, early delivery
Applicability	Best for code correctness and reliability	Ideal for complex requirements and stakeholder involvement	Suitable for projects with well-defined features and milestone planning

Test-Driven Development (TDD)	Behavior-Driven Development (BDD)	Feature-Driven Development (FDD)
<ul style="list-style-type: none"><li>● Approach: Write tests first.</li><li>● Benefits: Early bug detection, and improved code quality.</li><li>● Suitability: Best for code correctness.</li></ul>	<ul style="list-style-type: none"><li>● Approach: Describe behavior before coding.</li><li>● Benefits: Clear communication, and business alignment.</li></ul>	<ul style="list-style-type: none"><li>● Approach: Develop based on features.</li><li>● Benefits: Structured development, early delivery.</li><li>● Suitability: Suitable for well-defined features.</li></ul>