Note on the Software Development Life Cycle (SDLC)

The **Software Development Life Cycle (SDLC)** is a step-by-step method that helps developers and teams plan, build, test, and maintain software in an organized and reliable way. While there are many ways to structure the SDLC (like Waterfall, Agile, etc.), the core idea remains the same: break the process into **phases** so you can tackle problems methodically.

Here are the **typical SDLC phases** (in a simplified form):

1. **Requirements & Planning**
    - **Goal**: Decide what your software should do and why.
    - **Activities**:
        - Talk to stakeholders (e.g., Employees, managers, investors, customers, SME) to identify what the program must accomplish.
        - List out features and constraints (time, tools, difficulty level).
        - Make a quick plan outlining the tasks, who will do them (if working in pairs/teams), and the rough timeline.
2. **Design**
    - **Goal**: Turn your ideas into a logical blueprint or plan.
    - **Activities**:
        - Sketch out the program flow using **pseudocode** or **flowcharts**.
        - Decide on data structures and how you'll handle user input and output (e.g., simple console input/output).
        - Note and design architecture (Project structure)
3. **Implementation (Coding)**
    - **Goal**: Write the actual code that follows your plan.
    - **Activities**:
        - Use meaningful variable names and basic commenting.
        - If you're new to coding, write small chunks at a time, then test them immediately.
        - Keep it simple: focus on **one feature** or **one step** before moving on.
4. **Testing**
    - **Goal**: Ensure the program works correctly and fix any mistakes.
    - **Activities**:
        - Run the program with typical and unexpected inputs to see if it behaves as intended.
        - Listen to feedback from classmates who try your program.
        - Fix errors (bugs) as you find them.
5. **Deployment**
    - **Goal**: Make your program available to use.
    - **Activities**:
        - If it's a console-based program, just make sure it can run on a classroom computer.
        - Provide a short note or instructions on how to start it (e.g., "Run this file in Python" or "Double-click the .exe").

6. **Maintenance**
    - **Goal**: Keep the software updated and working.
    - **Activities**:
        - If someone reports a new bug or needs a small feature change, update the code.
        - Ensure your documentation (comments, instructions) remains accurate.

Why the SDLC Matters, Even for Small Projects

- **Organization**: It keeps you from diving straight into code without a plan.
- **Quality**: You can catch problems early by writing and testing code in small steps.
- **Clarity**: It's easier to explain your code to colleagues (or yourself, weeks later!).
- **Less Stress**: Structured steps help you manage time, avoid confusion, and create better projects.

Fun Lab: "Two-Question Multiple-Choice Quiz"

Overview

- **Project**: A very simple, **console-based quiz** that asks the user **two multiple-choice questions** and provides a score at the end.
- **Goal**: Learn and apply each **SDLC phase** in a single day (1–2 hours of class or a short, dedicated block).

Phase 1: Requirements & Planning (10–15 minutes)

1. **Requirements**
    - The quiz must ask **two** different multiple-choice questions.
    - The quiz should accept user input (e.g., "A", "B", "C", or "D").
    - Display how many questions the user got correct out of 2.
2. **Scope**
    - Keep it **console-based** (no fancy graphics).
    - Only two questions, each with 3 or 4 possible answers.
3. **Success Criteria**
    - The quiz runs without errors.
    - The user's score (0, 1, or 2 correct) is shown at the end.

Deliverable: A short list of features on paper or a whiteboard.

Phase 2: Design (10–15 minutes)

1. **Pseudocode**

Example structure:

```
START
Print "Welcome to the Quiz!"
Set score = 0

Print "Question 1: [Your question]"
Print "A) [Option A]"
Print "B) [Option B]"
Print "C) [Option C]"
...
Get userAnswer1
If userAnswer1 is correct
  score = score + 1

Print "Question 2: [Your second question]"
...
If userAnswer2 is correct
  score = score + 1

Print "Your final score is: " + score + "/2"
END
```

   -
2. **Data Structures**
   - Keep it simple: maybe just variables like score, userAnswer1, userAnswer2.
   - No advanced data structures needed for just two questions.

Deliverable: A half-page of pseudocode or a brief flowchart.

Phase 3: Implementation (Coding) (30–45 minutes)

1. **Set Up the Project**
   - Create a new file (e.g., TwoQuestionQuiz.py or TwoQuestionQuiz.java).
   - Initialize variables: score = 0.
2. **Write the Quiz**
   - Print your first question and answers (A, B, C, D).
   - Read the user's input.
   - Compare it to the correct answer; if correct, increment score.
   - Repeat for the second question.
3. **Basic Error Handling** (Optional)

- If the user types something invalid, you could display a message like “Invalid answer, skipping!”

Deliverable: A **working code** file you can run from a console or IDE.

Phase 4: Testing (15–20 minutes)

1. **Test Valid Answers**
   - Input “A”, “B”, “C”, or “D” to ensure the program acknowledges correct/incorrect answers properly.
2. **Edge Cases**
   - What if the user inputs lowercase letters or something else (like “X”)?
   - Decide if you want to handle it or just display “That’s incorrect.”
3. **Peer Testing**
   - Swap with a classmate—ask them to run your quiz and see if everything works.
   - Collect feedback (“It would be nice if it said ‘Good job!’ when correct.”).

Deliverable: Make any quick fixes based on test results.

Phase 5: Deployment (5 minutes)

1. **Share Your Quiz**
   - Show your teacher or classmates how to run it (“Open it in Python/Java and press ‘Run’”).
   - No advanced packaging needed for a simple console program.
2. **Final Check**
   - Confirm that the program starts up and ends gracefully.
   - Display the correct final score each time.

Deliverable: A final quiz file that runs without issues.

Phase 6: Maintenance (as needed)

1. **Optional Improvements**
   - Add more questions if time remains.
   - Add randomization of questions or a timer if you want a challenge.
2. **Bug Fixes**
   - If classmates find any issues (typos, wrong scoring), fix them quickly.

Deliverable: If you make changes, you’ll have an improved quiz ready for more testing.

No-Code, One-Day Lab: “Dream Up a Project”

Lab Overview

In this lab, students will **invent** a software (or system) concept and walk through the **SDLC phases** without writing a single line of code. The result will be a **high-level project outline** that demonstrates how each phase of the SDLC informs the next.

Estimated Time: 1 day (or a single class block of 1–2 hours)

Phase 1: Requirements & Planning (15–20 minutes)

1. **Brainstorm Ideas**
    - Individually or in small groups, think of a software or digital service you wish existed. Examples:
        - A scheduling app that automatically syncs with your teachers’ assignment calendars.
        - A “study buddy” website that randomly pairs students for revision.
        - A virtual pet app that encourages real-world exercise.
    - **Choose one** idea to develop during the rest of the lab.
2. **Define Requirements**
    - **Functional Requirements**: List out the main tasks your solution will perform. (e.g., “Automatically reminds users of homework deadlines.”)
    - **Non-functional Requirements**: Performance, usability, or security considerations. (Keep these simple, e.g., “App must be easy to use on a phone.”)
3. **Basic Plan**
    - Decide the **scope**: Are you designing a mobile app, a website, or a simple desktop concept?
    - Consider constraints: Time, skill level, or hypothetical budgets.
    - Outline who’s doing what if you’re working in pairs.

Deliverable: A short **“Idea & Requirements”** document (a few bullet points).

Phase 2: Design (15–20 minutes)

1. **Create a Flowchart or Concept Map**
    - Show how a user would navigate your system. For example:
        - **Start → Sign In → View Dashboard → Use Key Feature → Log Out**
    - Label steps with brief notes about what happens in each.

2. **Optional Wireframes/Sketches**
    - If your idea is visually oriented (e.g., a mobile app), sketch simple **wireframes** of the main screens or pages.
    - Keep it basic: Boxes for images, text placeholders for content, arrows showing navigation.
3. **Data Considerations**
    - If relevant, note how you'd store or track data. (For example, "Users' assignment details are stored in a spreadsheet or small database.")

Deliverable:

- A **flowchart** or **concept map**
- (Optional) **Wireframes** or sketches of main screens/pages

---

Phase 3: Implementation (Conceptual, 20–30 minutes)

1. **Detail the "Building" Steps (No Code)**
    - Explain how you'd "construct" your app, site, or system.
    - For example:
        - "We'd use a simple web framework like HTML/CSS/JavaScript for the front end."
        - "We'd store data in a small SQL or NoSQL database."
        - If your project is just conceptual, say: "We'd hire a developer or use a drag-and-drop app builder."
2. **Outline Key Components**
    - For a minimal app, you might have:
        - **Front-end**: The user interface.
        - **Back-end**: The logic or "brain" of the system.
        - **Database**: Where data is stored.
    - If your idea is simpler (maybe just a website), keep the "components" to a website homepage, sub-pages, and a contact form.

Deliverable: A **written overview** (bullet points) describing how you'd build the product if you had the time and resources.

Phase 4: Testing (15–20 minutes)

1. **Hypothetical Test Cases**
    - Dream up scenarios you'd test if your system were built:
        - **Normal Use**: "User signs in, sees all their tasks."
        - **Error Cases**: "User enters the wrong password."
        - **Edge Cases**: "Database is temporarily unavailable."
    - How would you respond or fix these issues?
2. **Peer Feedback**

    - Trade your designs with another group.
    - Have them evaluate the flowchart or wireframes: “Would you know how to use it if it existed?” “Does it meet the stated requirements?”
3. **Refinements**
    - Make notes on what changes you’d make after hearing the feedback.

Deliverable: A **short test plan** (a few bullet points) plus any feedback notes you received.

Phase 5: Deployment (5–10 minutes)

1. **How Would You Release It?**
    - If it’s a mobile app, you might put it on the App Store or Google Play.
    - If it’s a website, you’d host it on a platform like GitHub Pages or a web hosting service.
    - If it’s an internal school tool, you’d share it on the school network or a class website.
2. **Consider a Simple User Guide**
    - Outline how someone would install or access it.
    - Provide a short “getting started” description for brand-new users.

Deliverable: A **deployment note** describing where and how your project would go live, plus a mini user guide (if you wish).

Phase 6: Maintenance (5–10 minutes)

1. **Post-Launch Plan**
    - How would you handle bug reports or feature requests?
    - Would you schedule monthly updates or wait until enough issues are reported?
2. **Future Enhancements**
    - Brainstorm one or two bigger ideas you’d add after the initial release.
        - For example, “Add a chat feature so users can message each other within the app.”

Deliverable: A **brief outline** of how you’d maintain and upgrade your project over time.

Conclusion

By **dreaming up** their own project and working through each phase of the **SDLC without coding**, grade 11 students learn to:

1. **Define** problems clearly,
2. **Design** solutions using flowcharts and wireframes,

3. **Conceptualize** how the final product might be built and tested,
4. **Plan** for deployment and future changes.