DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Lecture-1

Week 1 — Lessons 1 & 2

Software Life Cycle: Introduction to Software Engineering, The Waterfall Model, Prototyping & Agile/Incremental Models

Course Code: CSE-3203

Course Title: Software Requirement Specifications

Year: 3 | Semester: 2nd | Credits: 2.0

Textbooks: Pressman (2001) • Sommerville (2000) • Lamb (1988)

Table of Contents

1. Introduction to Software Engineering

1.1 What is Software?

Software is a collection of programs, data, and documentation that collectively make up a computer system. Unlike hardware, software is intangible — it cannot be touched, but its effects are felt everywhere. Understanding what software is, and why engineering principles are needed to build it, is the foundation of this course.

In everyday terms, we interact with software constantly: when we browse the internet, use a mobile app, or even access an ATM. Behind every interaction is a carefully constructed set of instructions running on a machine. The discipline of software engineering ensures those instructions are correct, efficient, maintainable, and delivered on time.

> **Software:** *A set of programs, procedures, and associated documentation concerned with the operation of a computer system. (Sommerville, 2000)*
>
> **Software Engineering:** *The application of a systematic, disciplined, and quantifiable approach to the development, operation, and maintenance of software. (IEEE Standard 610.12)*

Software engineering emerged as a formal discipline in the late 1960s precisely because building software without structure led to massive failures. Before software engineering was recognised, programmers worked informally, producing systems that were unreliable, costly to maintain, and almost impossible to scale.

1.2 Characteristics of Software

Software has several unique characteristics that distinguish it from traditional engineering products such as bridges or automobiles. Understanding these characteristics helps explain why software development is so challenging.

- **Software is developed or engineered — not manufactured.** Unlike physical products assembled on a factory floor, software is created through intellectual effort. This means quality must be designed in from the start, not inspected at the end.
- **Software does not wear out, but it deteriorates.** A physical product wears out due to friction and use. Software does not degrade physically, but it deteriorates over time through changes, patches, and modifications that introduce new bugs.
- **Software is complex.** Even a moderately sized software system involves millions of interdependent logical decisions. This complexity is orders of magnitude greater than in most other engineering disciplines.
- **Software is invisible.** You cannot see or touch software. Its structure and quality are hidden, making it difficult to assess progress or detect problems.
- **Software is flexible.** Software can theoretically be changed at any time. This flexibility is both an advantage and a major source of problems, as stakeholders often request changes without appreciating the cost.
- **Software is often custom-built.** Unlike hardware components, most software is built from scratch for a specific application or context, meaning accumulated experience rarely transfers perfectly.

> *The invisibility and complexity of software are two primary reasons why it is so difficult to manage. Unlike a construction project where the building's progress is visible, software defects may be hidden for months.*

1.3 The Software Crisis

The term 'software crisis' was coined at the 1968 NATO Software Engineering Conference in Garmisch, Germany. It described a period where the demand for software vastly outpaced the ability of the industry to deliver quality software reliably, on time, and within budget.

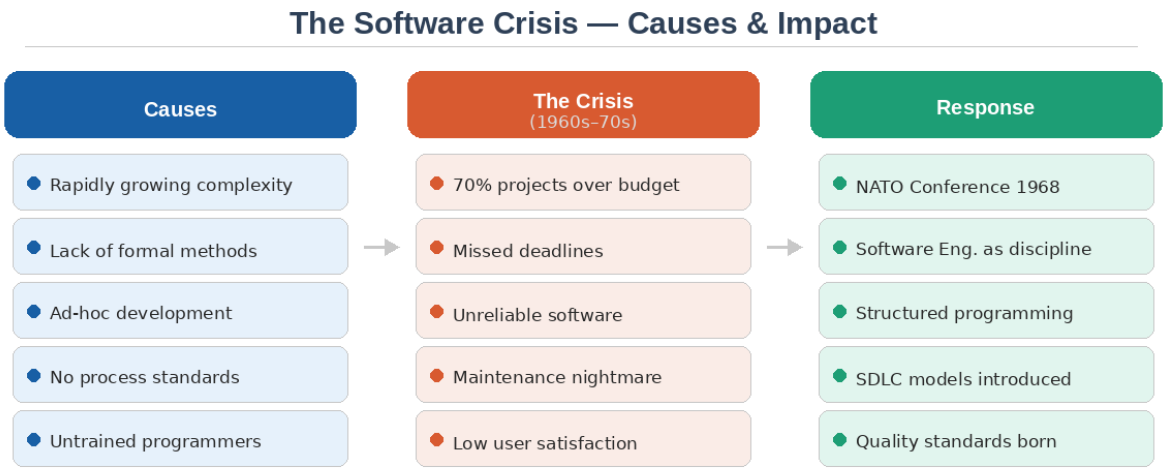

Figure: The Software Crisis — Causes, Symptoms, and Response

The symptoms of the software crisis were alarming. Studies from the era showed that the majority of large software projects either failed outright or delivered systems that were significantly over budget or behind schedule. The causes were rooted in a lack of formal methods, an absence of process standards, and the use of ad-hoc development practices.

- **Budget Overruns:** Studies showed over 70% of large software projects exceeded their budgets.
- **Schedule Delays:** Most large projects took two to three times longer than originally estimated.
- **Unreliable Delivery:** Software was often delivered with critical bugs and did not meet user requirements.
- **Maintenance Nightmare:** Once deployed, software was almost impossible to modify. Changes introduced new bugs faster than old ones were fixed.
- **No Formal Methods:** There were no standardised processes, tools, or notations. Development depended entirely on individual programmer skill.

The response to this crisis was the formal birth of software engineering. The 1968 NATO conference proposed treating software development with the same rigour applied to other engineering disciplines — with structured processes, formal notations, quality standards, and project management frameworks.

1.4 Goals of Software Engineering

Software engineering aims to produce software that satisfies a set of critical quality attributes. These goals guide every decision made during development:

- **Correctness:** The software must do exactly what the requirements specify. This is the most fundamental quality attribute.

- **Reliability:** The software must perform its intended function under all specified conditions without failure.
- **Efficiency:** The software must make optimal use of computing resources such as memory, processing time, and storage.
- **Maintainability:** The software must be easy to modify to correct defects, improve performance, or adapt to a changing environment.
- **Portability:** The software should be transferable from one environment to another with minimal effort.
- **Usability:** The software must be easy and intuitive for end users to operate.

> *These goals often conflict with each other. For example, maximising efficiency may reduce maintainability. Good software engineering involves making informed trade-offs.*

2. The Classical Life Cycle — The Waterfall Model

2.1 Overview and Phases

The Waterfall Model, proposed by Winston Royce in 1970, is the oldest and most widely known Software Development Life Cycle (SDLC) model. It organizes development into a sequence of well-defined phases, where each phase must be completed and formally reviewed before the next phase begins.

The name comes from the way progress flows steadily downward — like a waterfall — through successive phases. Each phase produces a deliverable that becomes the input to the next. This sequential nature makes the model easy to understand and manage, but also inflexible in the face of changing requirements.

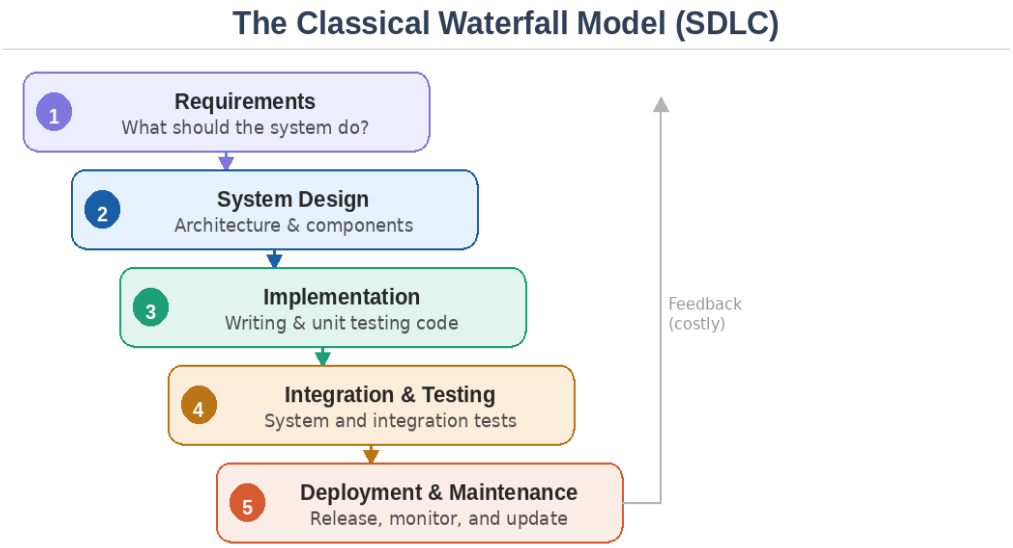

Figure: The Classical Waterfall Model — Five Sequential Phases

Phase 1: Requirements Analysis

In this phase, all requirements for the system are gathered from stakeholders and documented in a Software Requirements Specification (SRS) document. Both functional requirements (what the system should do) and non-functional requirements (performance, security, usability) are captured. This phase is critical because any errors here propagate through every subsequent phase.

Phase 2: System Design

Based on the requirements document, system architects design the overall system structure. This includes defining the hardware configuration, system architecture, network layout, database design, and the high-level design of each module. A key output is the Design Document Specification (DDS).

Phase 3: Implementation (Coding)

The system design is translated into source code. Developers write code in the chosen programming language, following the design specifications. Individual units or modules are also tested during this phase through unit testing.

Phase 4: Integration and Testing

All modules are integrated and tested as a complete system. Various testing strategies are applied — integration testing, system testing, performance testing, and acceptance testing. Defects found here may require changes that cascade back through earlier phases.

Phase 5: Deployment and Maintenance

The finished, tested product is deployed to the production environment. This phase also includes all post-deployment activities: fixing discovered bugs, adapting the software to new hardware or operating system environments, and enhancing features based on user feedback. Maintenance typically consumes 60–70% of a software product's total lifecycle cost.

2.2 Advantages and Limitations

The Waterfall model's straightforward structure gives it both strengths and significant weaknesses. The following table summarises both sides:

Advantages	Limitations
Simple and easy to understand	Very inflexible to requirement changes
Well-defined milestones per phase	No working software until very late
Easy to manage (rigid structure)	High risk for complex or long projects
Works well for stable requirements	Poor model when requirements are unclear
Clear documentation at every stage	Expensive to go back and fix earlier phases

2.3 When to Use the Waterfall Model

Despite its limitations, the Waterfall model remains relevant for certain types of projects. It is most appropriate when:

- **Requirements are very well understood, stable, and unlikely to change during development.**
- **The project is short-duration with a clearly scoped deliverable.**
- **The technology is mature and well-understood by the development team.**
- **The customer does not require frequent demonstrations or intermediate deliverables.**
- **The team is large and geographically distributed, requiring rigid process controls.**

Typical real-world examples include government contracts, defence systems, and infrastructure software where requirements are locked in by legal or contractual agreements before development begins.

> *The Waterfall model is not suitable for most modern commercial software projects where requirements evolve rapidly, competitive pressure demands early delivery, and user feedback is essential.*

3. The Prototyping Model

3.1 Concept and Purpose

The Prototyping Model was developed to directly address the most significant weakness of the Waterfall model: the assumption that requirements can be fully and accurately specified before development begins. In practice, users often struggle to articulate their needs precisely until they see a working system.

Prototyping solves this by building a quick, partial version of the system early on. This prototype is demonstrated to stakeholders, who provide feedback that is used to refine requirements. The cycle repeats until the requirements are clear enough to build the final system.

The core philosophy of prototyping is: build to learn, then build to deliver. The prototype is an exploratory tool, not the final product. It answers the question: 'Is this what you want?' before significant resources are committed to full-scale development.

> **Prototype:** *A preliminary model or working simulation of the final system, built quickly and cheaply to elicit feedback and clarify requirements.*

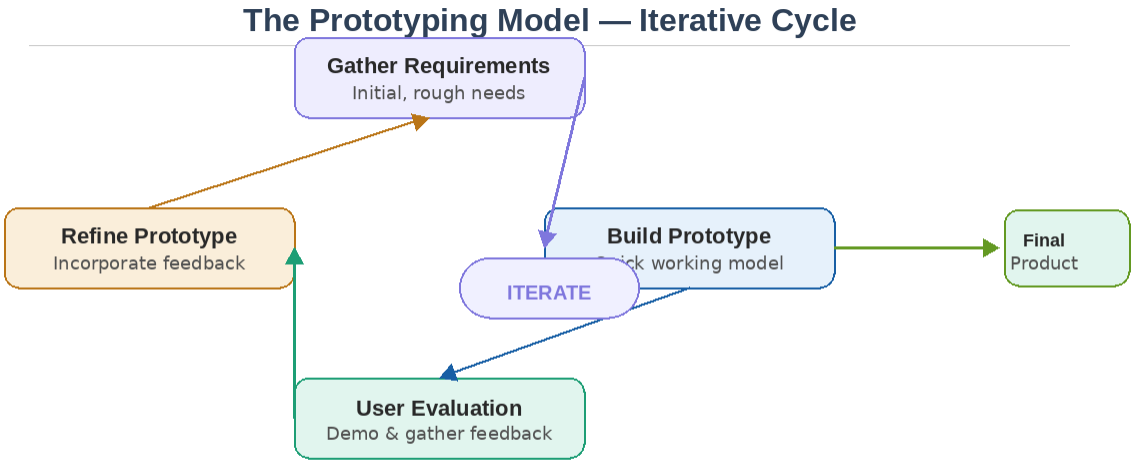

Figure: The Prototyping Model — Iterative Requirements Cycle

3.2 Types of Prototypes

Prototypes can be built in different ways depending on the purpose of the prototype and the type of feedback sought. The four main types are:

Type	Description	Best Used When
Throwaway	Built quickly to explore requirements; discarded after evaluation	Requirements are ambiguous and need clarification
Evolutionary	Refined iteratively; becomes the final system	Long-term projects with changing requirements

Horizontal	Full interface, limited functionality; shows UI breadth	UI is the primary concern; stakeholder demos
Vertical	Deep implementation of a narrow system slice	Testing feasibility of a technical component

3.3 The Prototyping Process

The prototyping process follows a clearly defined cycle that repeats until both the developer and the user are satisfied that requirements are fully understood:

- **Step 1 — Gather Initial Requirements:** A rough, initial set of requirements is collected from the customer. These need not be complete or precise — the prototype will clarify them.
- **Step 2 — Build the Prototype:** A simplified version of the system is constructed rapidly. The focus is on visible features — the user interface and key workflows — rather than full backend functionality.
- **Step 3 — Evaluate with the Customer:** The prototype is demonstrated to the customer in a structured walkthrough. The customer identifies what is correct, what is missing, and what needs to change.
- **Step 4 — Revise and Refine:** Based on the customer's feedback, the prototype is refined. In some cases, the underlying design changes significantly.
- **Step 5 — Repeat:** Steps 3 and 4 repeat until the customer approves the prototype as an accurate representation of requirements.
- **Step 6 — Engineer the Final System:** The agreed requirements are used as the basis for building the actual, production-quality system.

> *A key risk with prototyping is that customers sometimes mistake the prototype for the finished product, leading to false expectations about delivery timelines and quality.*

3.4 Advantages and Disadvantages

Advantages

- **Reduces requirements uncertainty by involving users early and often.**
- **Misunderstandings between users and developers are identified and corrected early, reducing costly late-stage changes.**
- **Users gain hands-on experience with the system's look and feel before full development begins.**
- **Supports innovation — developers can experiment with interface designs and workflows.**
- **Works well for large, complex systems where requirements are inherently difficult to specify upfront.**

Disadvantages

- **Customers may confuse the prototype with the final product, leading to scope creep and unrealistic expectations.**
- **Developers may compromise on design quality to rapidly build the prototype, and these shortcuts may be retained in the final system.**

- **The process can become open-ended if there is no formal agreement on when prototyping ends and final development begins.**
- **Documentation can be neglected during rapid prototyping cycles, creating maintenance problems later.**

4. Agile and Incremental Models

4.1 The Incremental Model

The Incremental Model combines the linear structure of the Waterfall model with the iterative idea of building the system in parts. Rather than delivering the entire system at once, the system is divided into smaller, independently deliverable increments. Each increment goes through its own mini-waterfall: requirements, design, implementation, and testing.

The first increment usually contains the core functionality — the features without which the system cannot operate. Subsequent increments add new features on top of the working foundation. Users receive a working, testable system early in the development cycle and provide feedback that shapes subsequent increments.

Key Characteristics of the Incremental Model

- **The system is broken into small, self-contained modules or builds.**
- **Each increment is a fully tested, working slice of the final system.**
- **Customers can use and evaluate earlier increments while later increments are being built.**
- **Requirements for later increments can be refined based on experience with earlier ones.**
- **Risk is reduced because problems in one increment do not jeopardise the entire project.**

When to Use the Incremental Model

- **Most requirements are understood but some may change over time.**
- **There is a need to deliver some functionality quickly (e.g. business demand for a first release).**
- **The system can be logically divided into semi-independent modules.**
- **Resources (staff, budget) are limited and must be allocated incrementally.**

4.2 The Agile Model

Agile is not a single process model but a philosophy of software development that emerged in the 1990s and was formalized by the Agile Manifesto in 2001. It represents a fundamental shift away from heavyweight, documentation-driven processes towards adaptive, people-centered development.

The Agile Manifesto defines four core values that guide all Agile methods:

- **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 frameworks such as Scrum, Extreme Programming (XP), and Kanban implement these values through specific roles, ceremonies, and artefacts. The unifying concept across all Agile frameworks is short, time-boxed development cycles — known as iterations or sprints — at the end of which working software is demonstrated to stakeholders.

> **Sprint:** *A time-boxed iteration in Agile (typically 2–4 weeks) at the end of which a potentially shippable product increment is produced.*
>
> **Scrum:** *The most widely used Agile framework, featuring defined roles (Product Owner, Scrum Master, Development Team), ceremonies (Sprint Planning, Daily Standup, Sprint Review, Sprint Retrospective), and artefacts (Product Backlog, Sprint Backlog, Increment).*

4.3 The Agile Sprint Cycle

The Sprint Cycle is the heartbeat of Agile development. Every activity in Agile is organised around this repeating cycle. The following diagram illustrates the five key stages of an Agile sprint:

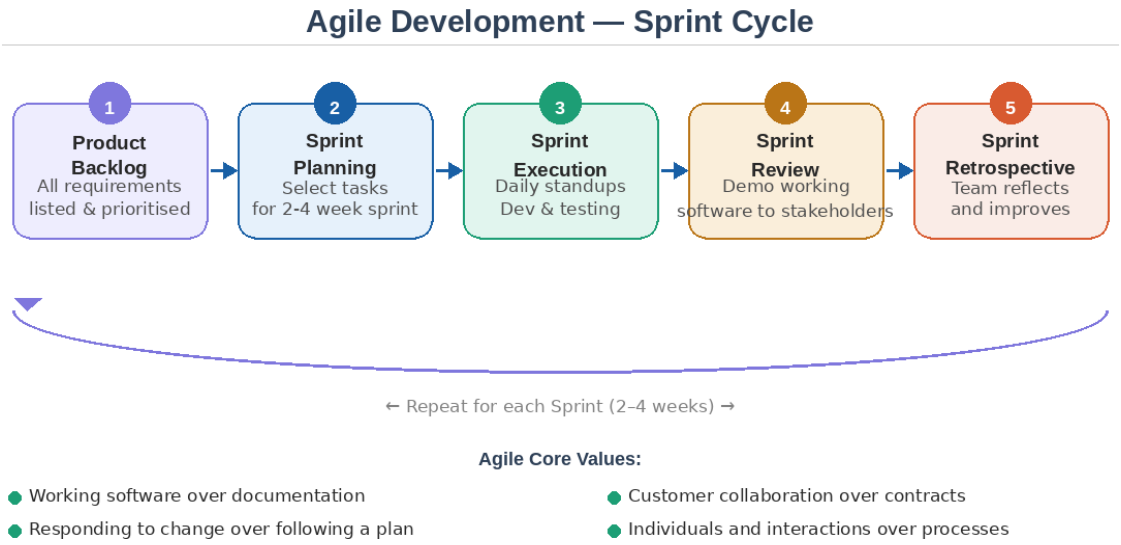

Figure: Agile Development — The Sprint Cycle

- **Product Backlog:** A prioritised list of all desired features, user stories, and bug fixes for the product. The Product Owner is responsible for maintaining and prioritising this list.
- **Sprint Planning:** The team selects a set of items from the product backlog to complete within the upcoming sprint. Effort is estimated and tasks are assigned.
- **Sprint Execution:** The development team builds, tests, and integrates the selected features. Daily standups (15-minute meetings) keep the team aligned and surface blockers quickly.
- **Sprint Review:** At the end of the sprint, the team demonstrates the working increment to stakeholders and collects feedback. The product backlog is updated accordingly.
- **Sprint Retrospective:** The team reflects on the process — what went well, what didn't, and what to improve in the next sprint. This is Agile's built-in continuous improvement mechanism.

> *In Agile, the guiding principle is to fail fast and learn fast. If a feature doesn't work or isn't what the customer wanted, it is discovered within weeks — not after years of development.*

5. Comparison of SDLC Models

The following table provides a comprehensive comparison of all four SDLC models covered in Week 1. Use this table as a quick reference to understand which model is most appropriate for a given project context.

Feature	Waterfall	Prototyping	Incremental	Agile
Flexibility	Low	Medium	Medium-High	Very High
Customer Involvement	Start & End	Per Prototype	Per Increment	Continuous
Delivery	Single release	Demo early	Multiple builds	Every sprint
Documentation	Very heavy	Moderate	Partial	Minimal
Best For	Stable requirements	Unclear requirements	Modular systems	Evolving requirements
Risk Level	High (late discovery)	Low (early feedback)	Medium	Low (rapid cycles)

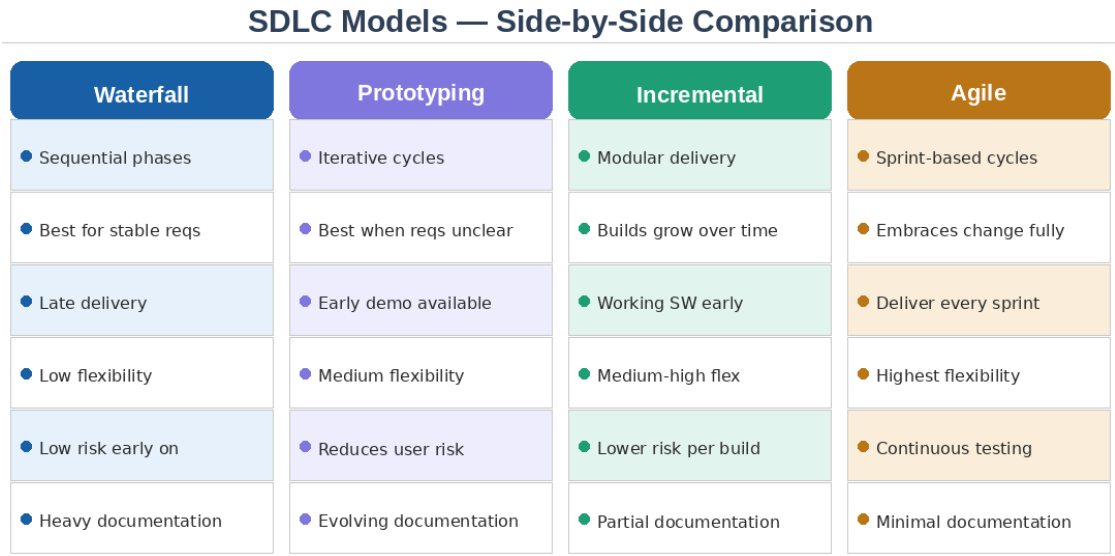

Figure: SDLC Models — Side-by-Side Comparison of Key Attributes

No single model is universally superior. The choice of SDLC model depends on the nature of the project, the clarity of requirements, team size, customer involvement, and risk tolerance. In practice, many modern teams adopt hybrid approaches — for example, using a Waterfall-like structure for high-level planning while using Agile sprints for implementation.

6. Key Takeaways and Review Questions

6.1 Key Takeaways

- **Software engineering emerged from the software crisis of the 1960s as a formal discipline to bring structure, quality, and predictability to software development.**
- **Software has unique characteristics — it is developed, not manufactured; it does not wear out but deteriorates through change; and it is invisible and complex.**
- **The Waterfall Model is the oldest SDLC model. It is sequential, easy to manage, and best for projects with stable requirements, but inflexible and high-risk for complex projects.**
- **The Prototyping Model reduces requirements risk by building early working models for user feedback. It is especially useful when requirements are unclear.**
- **Incremental and Agile models deliver software in small, working pieces, enabling early value delivery, rapid feedback, and adaptability to change.**
- **No SDLC model is perfect — each involves trade-offs between flexibility, structure, speed, and documentation.**

6.2 Key Definitions to Remember

> **Software Engineering:** *Application of systematic, disciplined, quantifiable approaches to software development, operation, and maintenance.*
>
> **SDLC:** *Software Development Life Cycle — the structured process used to plan, design, develop, test, and maintain a software system.*
>
> **Waterfall Model:** *A sequential SDLC model where each phase must be completed before the next begins; best for stable requirements.*
>
> **Prototype:** *An early working model of a system built to clarify requirements and gather user feedback before full development.*
>
> **Agile:** *A philosophy and set of practices that prioritise adaptive planning, early delivery, and continuous improvement through short iterative cycles.*
>
> **Sprint:** *A time-boxed iteration in Agile (2–4 weeks) that produces a potentially shippable product increment.*

6.3 Review Questions

Answer the following questions to test your understanding of the Week 1 material:

Q1. Define software engineering and explain why it emerged as a formal discipline in the late 1960s.

Q2. List and briefly describe five key characteristics that distinguish software from physical engineering products.

Q3. Draw and label the five phases of the Waterfall model. What are its two main limitations?

Q4. Explain the difference between a throwaway prototype and an evolutionary prototype. Give one scenario where each would be most appropriate.

Q5. What are the four core values of the Agile Manifesto? How do they contrast with the Waterfall approach?

Q6. Compare the Incremental Model and the Agile Model. In what key ways are they similar? In what ways do they differ?

Q7. A company wants to build an air traffic control system. Requirements are fixed by safety regulations and will not change. Which SDLC model would you recommend? Justify your answer.

Q8. A startup is building a new social media app. Features are likely to change based on user feedback. Which SDLC model would you recommend? Why?

6.4 Recommended Readings

- **Pressman, R.S. —** Software Engineering: A Practitioner's Approach (McGraw-Hill, 2001) — Chapters 1 and 2
- **Sommerville, I. —** Software Engineering, 5th Edition (Addison-Wesley, 2000) — Chapters 1 and 3
- **Lamb, D.A. —** Software Engineering (Prentice-Hall, 1988) — Chapter 1: Introduction

End of Week 1 Class Notes — CSE-3203