**Unit-1 • Software and Models**

**Lecture Hours: 15**

**0) Big picture: What this unit covers**

* What **software** is, its **characteristics**, **components**, **applications**, and a crisp **definition**.
* Major **software process models**: **Waterfall**, **Spiral**, **Prototyping**.
* Core **project management** ideas you actually use on projects.
* Why **metrics & measurements** matter and the most common ones.
* **Self-study**: **Fourth Generation Techniques (4GT)**.

**1) Software**

**1.1 Definition (exam-safe)**

**Software** is a set of **computer programs**, **data**, and **documentation** that instructs a computer to perform tasks and provides services to users.

* **Programs** = instructions (code).
* **Data** = what programs read/write (files, DB rows, images).
* **Documentation** = how to install, use, and maintain.

**Why this matters:** hardware without software is like a phone with no apps—powerful but idle.

**1.2 Characteristics of software (with plain-English notes)**

1. **Intangible** – you can’t touch it; you only *execute* it.
2. **Developed, not manufactured** – made by design + coding; no factory line.
3. **Doesn’t wear out** – no rust; but it **ages** when requirements change (needs updates).
4. **Custom-built and complex** – many interacting parts; tiny mistakes → bugs.
5. **Highly changeable** – business, laws, or users change → software must evolve.
6. **Quality depends on process** – good methods (reviews, tests) → fewer defects.
7. **Human-intensive** – main cost is people’s time, not materials.
8. **Error-prone** – invisible logic; testing/review is vital.

**1.3 Components of software**

1. **Programs (Code)** – functions, classes, modules.
2. **Data** – configs, DB schemas, seed data, media.
3. **Documentation** – user guides, API docs, design docs, test plans.
4. **Configuration** – environment files, dependencies, build scripts.
5. **Supporting assets** – UI layouts, icons, localization files.

**Tip:** Exams often ask “programs + docs + data” — add **configuration** to earn extra credit.

**Applications of Software (Detailed)**

Software is used in **almost every field of human activity**. Below are the main categories of applications with detailed explanations.

**1) System Software**

👉 System software acts as a platform for other software. It directly controls and manages computer hardware.

**Components:**

1. **Operating Systems (OS)**
   * Examples: Windows, Linux, macOS, Android, iOS.
   * Functions: manage CPU, memory, files, input/output devices, user interface.
   * Use-case: Without OS, you cannot run apps or connect devices.
2. **Device Drivers**
   * Bridge between OS and hardware.
   * Example: Printer driver, graphics driver, sound driver.
   * Use-case: A printer won’t work unless its driver is installed.
3. **Utility Programs**
   * Tools for system maintenance and optimization.
   * Examples: Antivirus, Disk Defragmenter, Backup tools, File Compression (WinRAR, 7-Zip).
   * Use-case: Improve performance, remove malware, manage files.

**2) Application Software**

👉 Directly helps users perform tasks.

**Categories:**

1. **General-purpose software**
   * Word processing: MS Word, Google Docs
   * Spreadsheets: MS Excel
   * Presentations: PowerPoint, Keynote
   * Web browsers: Chrome, Firefox
   * Email: Outlook, Thunderbird
   * Use-case: Office automation, personal productivity.
2. **Special-purpose software**
   * Focused on specific tasks.
   * Examples: Photoshop (image editing), AutoCAD (design), Tally (accounting), SPSS (statistics).
   * Use-case: Professionals use it in their field for efficiency.
3. **Business & Enterprise applications**
   * ERP (Enterprise Resource Planning): SAP, Oracle ERP
   * CRM (Customer Relationship Management): Salesforce, Zoho CRM
   * HRMS (Human Resource Management): Workday
   * Use-case: Helps large companies manage finance, sales, employees, supply chains.

**3) Embedded Software**

👉 Built into hardware devices to control specific functions. Usually invisible to users.

* Examples:
  + Car software for engine control and airbags.
  + ATM machines running banking software.
  + Washing machines, microwaves, smart TVs.
* Use-case: Automates device operations without requiring user programming.

**4) Web-based Software**

👉 Software that runs on the web through browsers, doesn’t need installation.

* Examples: Gmail, YouTube, Facebook, Amazon, Google Docs.
* Use-case: Online shopping, communication, entertainment, productivity.
* Features: Access from anywhere, multi-user, cloud storage.

**5) Mobile Applications**

👉 Apps built for smartphones and tablets.

* Examples: WhatsApp (chat), PayTM/Google Pay (payments), Instagram (social media), Zomato/Swiggy (food delivery).
* Use-case: On-the-go communication, e-commerce, banking, gaming.
* Importance: Makes services portable and user-friendly.

**2) Software Models (a.k.a. SDLC Process Models)**

**2.0 What is a process model?**

A **process model** is a **recipe** for building software—what phases happen, in what order, and with what outputs.

**Common phases (generic SDLC):**  
Requirements → Design → Coding → Testing → Deployment → Maintenance

**2.1 Waterfall Model**

**Idea:** Linear sequence. Finish one phase, then start the next.

**Phases & outputs**

1. **Requirements** → SRS (Software Requirements Specification)
2. **System/Design** → HLD (architecture), LLD (detailed design)
3. **Implementation** → Source code, unit tests
4. **Integration & Testing** → Test reports (system, acceptance)
5. **Deployment** → Release notes, user manual
6. **Maintenance** → patches, updates

**Strengths**

* Very **simple** and **well-documented**.
* Works when requirements are **stable** and **well-known**.
* Easy **milestone tracking**.

**Weaknesses**

* **Late testing** → issues discovered near the end.
* **Change-unfriendly** after early phases close.
* **Customer feedback** arrives very late.

**When to use**

* Regulations demand heavy documentation (e.g., defense).
* Short, well-understood projects (e.g., a calculator firmware).

**Mnemonic**: Water flows **down** the fall; so phases go **down** once.

**2.2 Spiral Model**

**Idea:** Build in **repeated cycles**, each cycle focuses on **risk analysis** first.

**Each spiral (cycle) has 4 quadrants**

1. **Objective setting & planning** (what to do this cycle)
2. **Risk assessment & reduction** (identify/mitigate risks)
3. **Development & validation** (build + test)
4. **Review & plan next cycle** (customer evaluation)

**Visual:** think of a spiral expanding outward—**radius = cost/time**, **angle = progress**.

**Strengths**

* **Risk-driven**: tackles high-risk parts early.
* Allows **customer feedback** each cycle.
* Good for **large, complex, mission-critical** systems.

**Weaknesses**

* Needs **experienced** risk managers.
* Can become **costly** and **long** if cycles are many.

**When to use**

* High uncertainty or high risk (safety, big budgets, evolving tech).

**2.3 Prototyping Model**

**Idea:** Build a **quick, limited version** (prototype) to learn what users want, then refine.

**Kinds of prototypes**

* **Throwaway (exploratory)**: discard after learning.
* **Evolutionary**: keep improving the same prototype into final product.
* **Incremental**: deliver features in increments.
* **Extreme (for web)**: UI-first mock → services → DB.

**Steps**  
Gather → Quick design → Build prototype → User evaluate → Refine → Final system

**Strengths**

* Reduces **requirement confusion**.
* Early **stakeholder visibility**.
* Better **UX** due to feedback.

**Weaknesses**

* Risk of **poor architecture** if prototype becomes final without redesign.
* Can **extend timeline** if constant changes.

**When to use**

* Requirements unclear; UI/UX heavy apps; start-ups/MVPs.

**3) Concepts of Project Management (software context)**

**3.1 What is a project?**

A **temporary** effort to create a **unique** product/service with **constraints**.

**The Triple Constraint (Iron Triangle)**

* **Scope** (what we build)
* **Time** (deadline)
* **Cost** (budget)  
  Quality sits in the middle—changing one side affects others.

**3.2 Project life cycle vs SDLC**

* **Project life cycle**: Initiation → Planning → Execution → Monitoring/Control → Closure
* **SDLC**: Requirements → Design → Code → Test → Deploy → Maintain

You manage **people/time/budget** in the project life cycle while delivering **software phases** in SDLC.

**3.3 Planning essentials**

* **Scope statement** & **WBS** (Work Breakdown Structure)
* **Estimation** (time/effort/cost): expert judgment, analogy, **COCOMO** basics
* **Scheduling**: **Gantt charts**, **PERT/CPM**, **critical path**
* **Resource plan**: team roles, tools, environments
* **Risk plan**: identify → analyze (probability × impact) → respond (avoid/mitigate/transfer/accept)
* **Quality plan**: standards, reviews, testing levels
* **Communication plan**: who needs what info, when

**3.4 Monitoring & control**

* **Status tracking** (done/blocked, burndown)
* **Variances**: Schedule Variance (SV), Cost Variance (CV)
* **Change control**: evaluate impact before approving scope changes
* **Configuration management**: version control, baselines

**3.5 Closure**

* Verify deliverables, lessons learned, final docs, support handover.

**Tiny numerical example (schedule variance)**

* Planned done by week 4 = 50%
* Actual done = 40%
* **SV = Actual − Planned = −10%** → you’re behind schedule.

**4) Role of Metrics & Measurements**

**4.1 Why measure?**

* **Predict** effort, cost, and time.
* **Control** quality and progress.
* **Improve** future projects with real data.

**4.2 Types of measures**

* **Direct**: countable (LOC, #defects).
* **Indirect**: inferred (maintainability, usability).

**4.3 Common product metrics**

* **LOC (Lines of Code)** – simple size measure.
* **Function Points (FP)** – size by **functionality** (inputs, outputs, files, interfaces).
* **Cyclomatic Complexity (V(G))** – decision complexity of code.
  + **Formula**: V(G) = **E − N + 2P**
    - E = edges, N = nodes in control flow graph, P = connected components (usually 1)
  + **Rule of thumb**: 1–10 OK, >10 needs refactor/tests.
* **Defect Density** = defects / KLOC (or /FP)
* **Code Coverage** (%) = covered lines / total lines × 100

**4.4 Process & project metrics**

* **Effort** (person-hours), **Productivity** (LOC/hour or FP/month)
* **Schedule Variance (SV)** = EV − PV
* **Cost Variance (CV)** = EV − AC
  + EV = Earned Value, PV = Planned Value, AC = Actual Cost
* **MTBF / MTTR**
  + **Mean Time Between Failures** (reliability)
  + **Mean Time To Repair** (maintainability)
* **Availability** = MTBF / (MTBF + MTTR)

**Mini worked example (defect density)**

* Found 24 defects in 12 KLOC → **Defect density = 24 / 12 = 2 defects/KLOC**.

**5) Self-Study: Fourth Generation Techniques (4GT)**

**5.1 What is 4GT?**

**Fourth Generation Techniques** aim to build apps with **very little coding** using **high-level, often visual or declarative tools**.

**5.2 Examples**

* **Non-procedural DB languages**: **SQL**, QBE (Query-By-Example)
* **Report generators**: Crystal Reports, JasperReports
* **GUI builders / RAD tools**: VB, Delphi, modern low-code/no-code platforms
* **Code generators**: ORM scaffolding, Swagger/OpenAPI client generators
* **CASE tools**: diagram → code stubs

**5.3 Pros / Cons**

**Pros**

* Very **fast** development; **business users** can participate.
* Less boilerplate; consistent patterns; fewer typos.

**Cons**

* **Performance** may be lower than hand-coded.
* **Lock-in** to a tool/platform.
* Hard to do **very custom** features.

**Where it shines**

* Admin dashboards, CRUD apps, reports, prototypes, small departmental apps.

**6) Putting it together (mini case)**

**Problem:** Student Information System (admissions, fees, results).

* If requirements are **fixed** (govt format), choose **Waterfall**.
* If stakeholders are **unsure about screens/flows**, start with **Prototyping** to refine UI.
* If there’s **high risk** (new tech, tight compliance), use **Spiral** to attack the risks early.
* Track **metrics** (defects per module, schedule variance) to stay on course.
* For quick admin reports, try **4GT** report builders.

**7) Typical exam questions (with short keys)**

1. **Define software. List its characteristics.**  
   *Programs + data + docs; intangible, developed not manufactured, doesn’t wear out, complex, changeable…*
2. **Explain Waterfall model with diagram and pros/cons.**  
   *Linear phases; simple but rigid; late testing…*
3. **Differentiate Spiral and Prototyping.**  
   *Spiral = risk-driven cycles; Prototyping = quick model for feedback.*
4. **What is project management? Explain triple constraint.**  
   *Scope–Time–Cost trade-off; quality impact.*
5. **What are software metrics? Explain LOC, FP, and Cyclomatic Complexity.**
6. **What are Fourth Generation Techniques? Give examples and advantages.**

**8) Ultra-short revision sheet (memorize)**

* **Software = Programs + Data + Docs**; intangible, complex, changeable.
* **Waterfall**: simple, fixed requirements; **Spiral**: risk-driven; **Prototype**: clarify requirements.
* **PM**: Plan → Schedule → Resource → Risk → Monitor → Close; **Scope–Time–Cost**.
* **Metrics**: LOC, FP, V(G)=E−N+2P, Defect/KLOC, SV, CV, MTBF/MTTR.
* **4GT**: SQL, report tools, low-code; **fast** but less flexible.