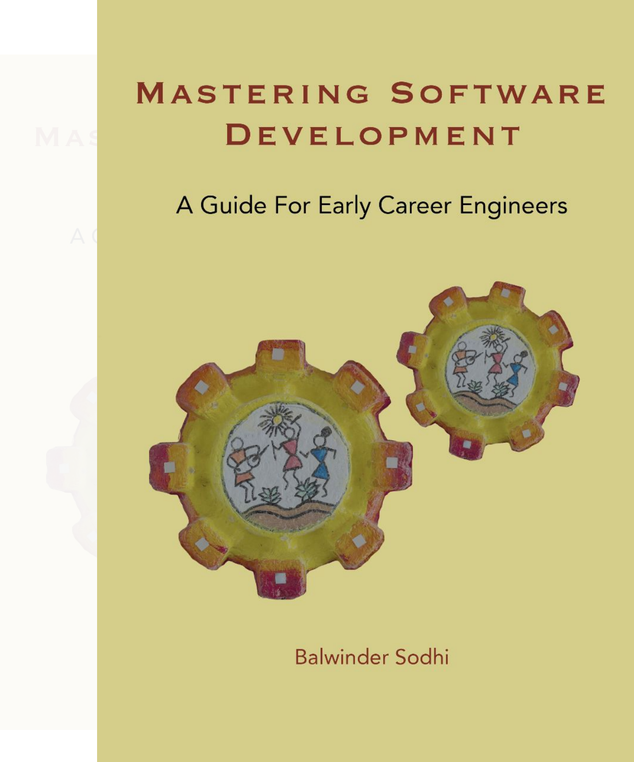


MASTERING SOFTWARE DEVELOPMENT

A Guide For Early Career Engineers

Balwinder Sodhi

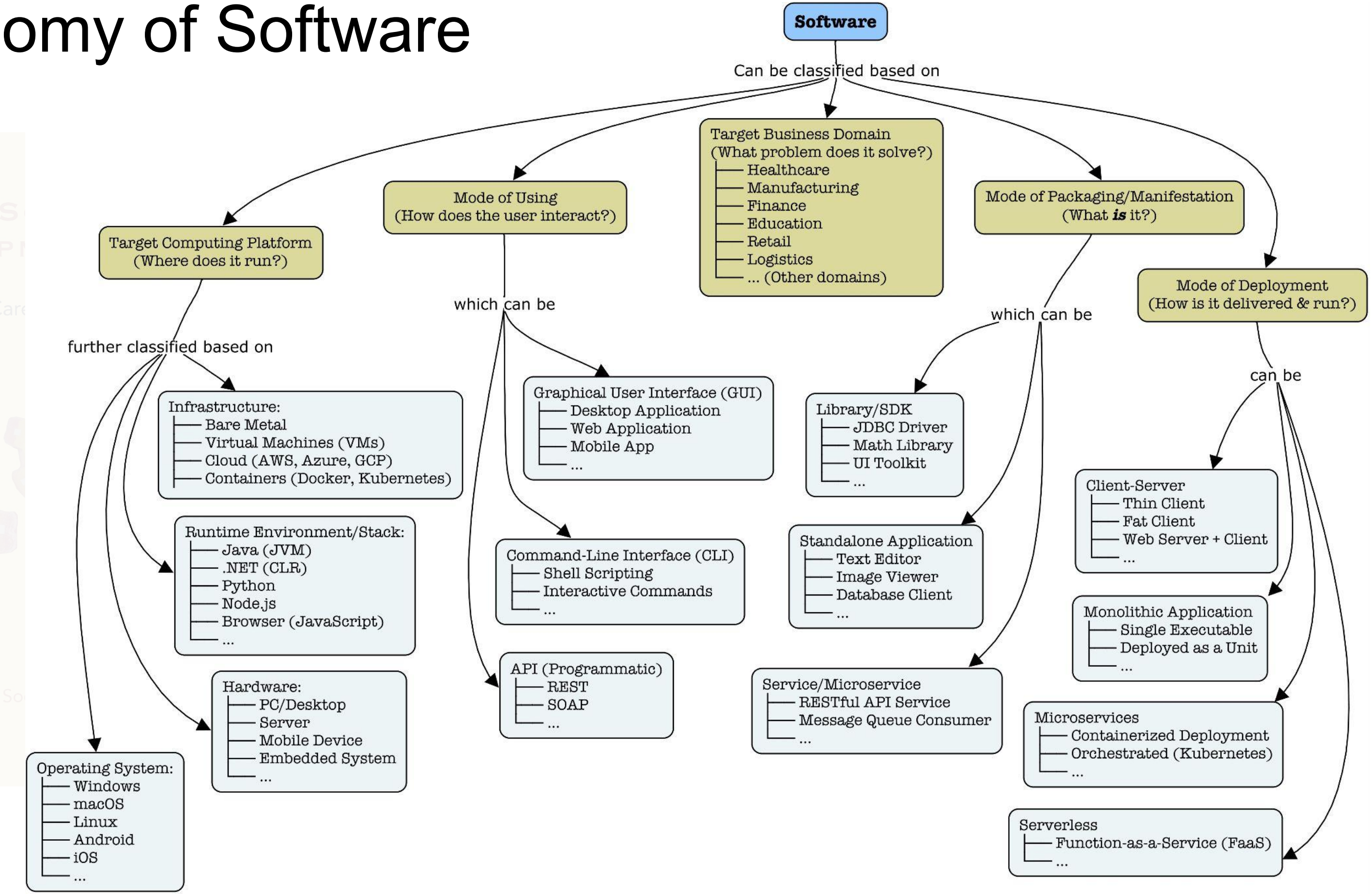
Foundations of Software Development

Balwinder Sodhi

Taxonomy of Software ─ Dimensions

- The target business domain
  - healthcare, manufacturing, finance, etc.
- The target computing platform for deployment or use of software
  - E.g., Java running on Linux, etc. on a PC, or virtual machines on a cloud data center, etc.
- Mode of using the software
  - E.g., CLI through a shell such as `ls`, `mkdir`, etc. or GUI tools etc.
- Mode of packaging or manifestation
  - E.g. as a library such as JDBC driver, or as a self-contained application such as a text editor.
- Mode of deployment
  - E.g., as a monolithic application, or as a remotely accessible client-server type of application.

Taxonomy of Software



Software

Can be classified based on

- Target Computing Platform (Where does it run?)
  - further classified based on
    - Infrastructure:
      - Bare Metal
      - Virtual Machines (VMs)
      - Cloud (AWS, Azure, GCP)
      - Containers (Docker, Kubernetes)
    - Runtime Environment/Stack:
      - Java (JVM)
      - .NET (CLR)
      - Python
      - Node.js
      - Browser (JavaScript)
      - ...
    - Hardware:
      - PC/Desktop
      - Server
      - Mobile Device
      - Embedded System
      - ...
    - Operating System:
      - Windows
      - macOS
      - Linux
      - Android
      - iOS
      - ...
- Mode of Using (How does the user interact?)
  - which can be
    - Graphical User Interface (GUI)
      - Desktop Application
      - Web Application
      - Mobile App
      - ...
    - Command-Line Interface (CLI)
      - Shell Scripting
      - Interactive Commands
      - ...
    - API (Programmatic)
      - REST
      - SOAP
      - ...
- Target Business Domain (What problem does it solve?)
  - Healthcare
  - Manufacturing
  - Finance
  - Education
  - Retail
  - Logistics
  - ... (Other domains)
- Mode of Packaging/Manifestation (What *is* it?)
  - which can be
    - Library/SDK
      - JDBC Driver
      - Math Library
      - UI Toolkit
      - ...
    - Standalone Application
      - Text Editor
      - Image Viewer
      - Database Client
      - ...
    - Service/Microservice
      - RESTful API Service
      - Message Queue Consumer
      - ...
- Mode of Deployment (How is it delivered & run?)
  - can be
    - Client-Server
      - Thin Client
      - Fat Client
      - Web Server + Client
      - ...
    - Monolithic Application
      - Single Executable
      - Deployed as a Unit
      - ...
    - Microservices
      - Containerized Deployment
      - Orchestrated (Kubernetes)
      - ...
    - Serverless
      - Function-as-a-Service (FaaS)
      - ...

Software Development Context

Mastering Software Development
A Guide For Early Career Engineers
Balwinder Sodhi

- Two broad contexts drive the nature of engineering decisions:
  - Building for internal business operations
  - Building as the product itself (customer-facing)
- This distinction affects:
  - Requirements
  - Quality expectations
  - Release cadence
  - Architecture decisions
  - Investment level

Building For a Business (Internal/Operational Software)

- Purpose: Automate workflows, improve efficiency, reduce costs.
- Examples: ERP customizations, logistics automation, internal analytics tools.
- Characteristics:
  - Requirements driven by internal stakeholders
  - Longer-lived systems with gradual evolution
  - Integration with legacy/internal systems
  - "Good enough" UX may be acceptable
  - Quality driven by reliability + maintainability
- Key Engineering Concerns:
  - Data correctness
  - Integration stability
  - Security & governance
  - Cost of change

Building As the Business (Core Product Software)

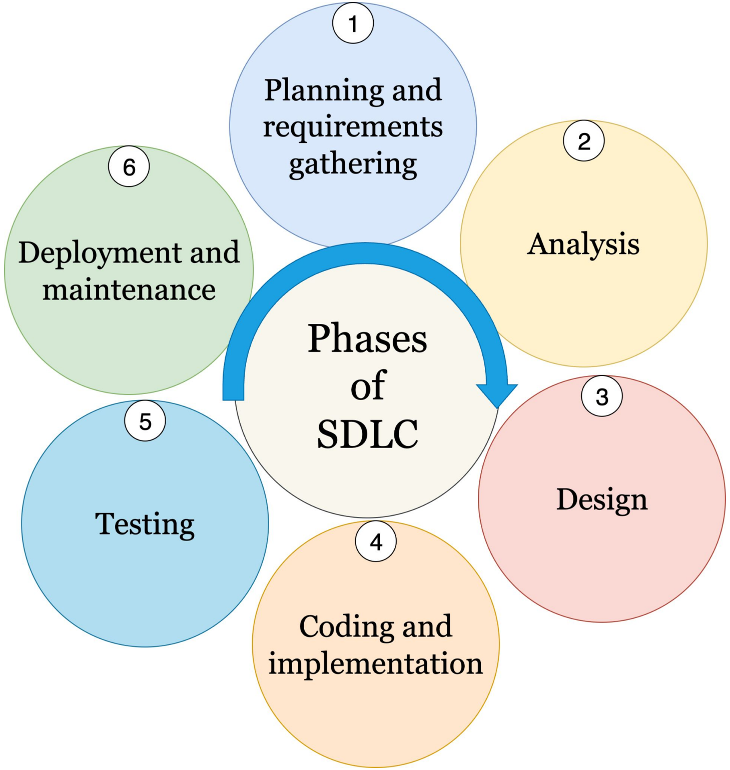
- Purpose: Software is the business’s core offering.
- Examples: SaaS platforms, fintech apps, marketplace systems, consumer apps.
- Characteristics:
  - Requirements driven by customers & market
  - UX and performance are differentiating factors
  - Feature velocity is critical
  - Higher competition & uptime expectations
  - Scales with user growth
- Key Engineering Concerns:
  - Scalability
  - Product-market fit iteration
  - Rapid delivery with quality
  - Observability & SLOs
  - Architecture flexibility

Software Development LifeCycle (SDLC)

- A structured process to build, deliver, and maintain software.
- Ensures:
    - Predictability
    - Quality
    - Manageability
    - Traceability
- Phases differ by model, but fundamentals remain consistent.

SDLC Phases

- Requirements Analysis
  - Problem definition, constraints, acceptance criteria.
- Design
  - Architecture, components, data design, interfaces.
- Implementation
  - Coding, code reviews, static analysis, version control.
- Testing
  - Unit, integration, system, performance, security.
- Deployment
  - Packaging, releasing, environment provisioning.
- Maintenance
  - Bug fixing, updates, optimizations, refactoring.


1
Planning and requirements gathering
2
Analysis
3
Design
4
Coding and implementation
5
Testing
6
Deployment and maintenance
Phases of SDLC

Common SDLC Models

Mastering Software Development
A Guide For Early Career Engineers
Balwinder Sodhi

- Waterfall
  - Linear and sequential.
- V-Model
  - Emphasizes verification/validation mapping.
- Iterative
  - Cycles of refinement.
- Incremental
  - Delivery in functional increments.
- Agile / Scrum / Kanban
  - Rapid iteration + customer collaboration.
- Spiral
  - Risk-driven layered development.
- DevOps-oriented SDLC
  - CI/CD, automation, rapid deployment.

Choosing the Right SDLC Model

- Depends on:
  - How fixed or evolving requirements are
  - Risk tolerance
  - Delivery timelines
  - Team size and maturity
  - Need for customer collaboration
  - Legacy constraints
  - Compliance requirements
- Typical heuristics:
  - Waterfall for stable, well-defined projects with compliance needs
  - Agile for evolving requirements and product-driven development
  - Iterative/Incremental for complex systems with staged growth
  - DevOps model when fast release cycles are essential

SDLC Documentation Overview

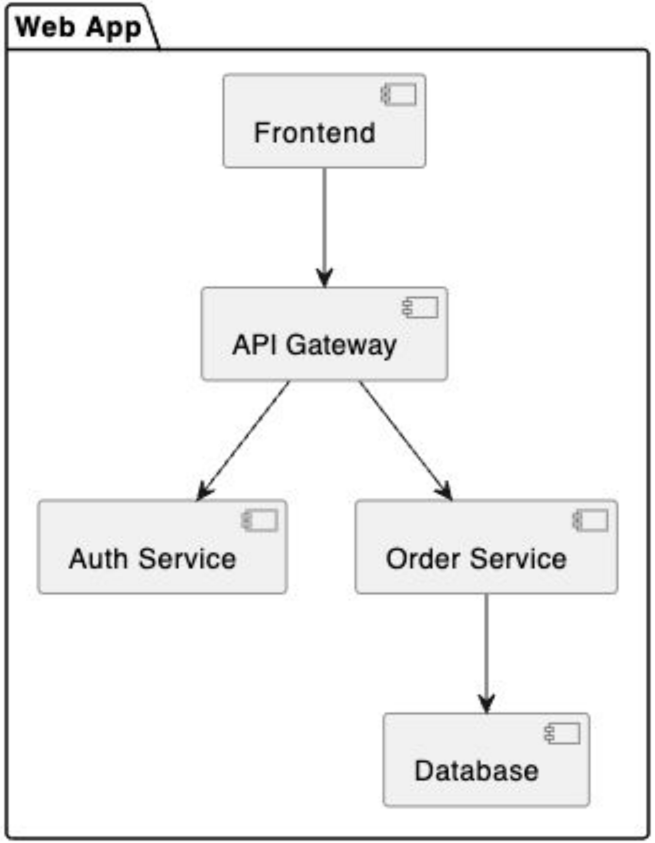
- Documentation supports clarity, alignment, and traceability.
- Includes:
  - ○ Requirements docs (SRS)
  - ○ Design docs (HLD/LLD)
  - ○ Test documentation
  - ○ User documentation
  - ○ API documentation

Software Requirements Specification (SRS)

- Purpose: Define what needs to be built.
- Contents:
  - Functional requirements
  - Non-functional requirements (performance, security)
  - Constraints
  - User stories / use cases
- Good SRS qualities:
  - Unambiguous
  - Verifiable
  - Complete
  - Feasible

Design Documents (HLD, LLD)

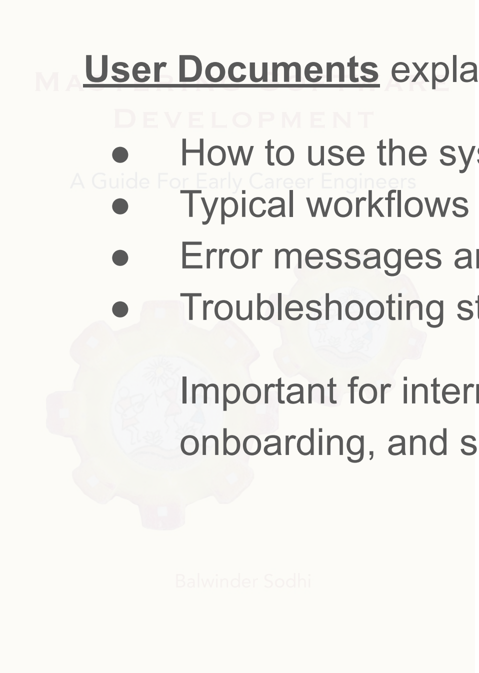
- High-Level Design (HLD):
  - ○ Architecture overview
  - ○ Subsystems/components
  - ○ High-level data models
  - ○ Technology choices
  - ○ Integration points
- Low-Level Design (LLD):
  - ○ Class-level details
  - ○ Data structures
  - ○ Algorithms
  - ○ API contracts
  - ○ Error-handling flows



Test Plans and Test Cases

- Test Plan Includes:
  - Scope
  - Test strategy
  - Tools
  - Environments
  - Roles/responsibilities
- Test Cases Include:
  - Preconditions
  - Test steps
  - Expected results
  - Pass/fail criteria
- Purpose: Ensuring full coverage and traceability back to requirements.

User-centric Documentation


Mastering Software Development
A Guide For Early Career Engineers
Balwinder Sodhi

User Documents explain:

- How to use the system
- Typical workflows
- Error messages and resolutions
- Troubleshooting steps

Important for internal adoption, customer onboarding, and support.

Good **API docs** include:

- Endpoint definitions
- Request/response formats
- Authentication details
- Error codes
- Example payloads
- Rate limits

Tools: OpenAPI/Swagger, Postman Collections, Redoc.

Challenges in SDLC Implementation

- Evolving Requirements
  - Changing business needs, market dynamics.
- Poor Communication
  - Misalignment between engineering, product, stakeholders.
- Low-quality Requirements
  - Ambiguous or incomplete specifications.
- Technical Debt
  - Accumulation slows delivery.
- Lack of Automation
  - Manual testing, deployments increase cycle time.
- Inadequate Documentation
  - Causes onboarding friction and maintenance issues.
- Insufficient Architecture Planning
  - Leads to scalability or reliability problems later.