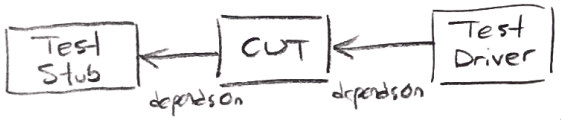
- Release Pipeline: Integration, Build, Deployment
- CI Loop: Commit, Build, Test, Report
- Gradle: Builds incrementally based on the task graph (DAG)
- SWE Activities: Specification, Development, Validation, Evolution
- User Requirements: Describe the services the system is expected to provide (High-Level)
- System Requirements: Detailed descriptions of system functions and operational constraints (low-level)
- Functional Requirements: Describe what the system should do "The system shall do..."
- NFRs: Specify properties of the system as a whole "The system shall be..."
- NFR Metrics: Speed, Reliability, Ease of Use, Size, Robustness, Portability
- Requirement Problems: Ambiguous, Incomplete, Inconsistent, Unquantifiable
- Good Requirements: Feasibility, Necessity, Testability
- User Stories: As a _, I want _ so that _
- Use Case: Title, Primary/Sec. Actors, Intention, Precondition(s), Main, Alternatives/Except., Post-condition(s)
- Every requirement needs at least 1 UC
- Each UC must be at least 1 business method
- Direct Type: No other types exist lower in the class hierarchy (vs. Indirect Type)
- Aggregation can be circular
- Foreign Key: The value in a column of Table 1 should be equal to some value of a specific column in Table 2
  → consistency + cascading deletes

- Object-Relational Mapping: Creates relational database tables from a domain model
- n-to-1 Associations:
  1) Add extra column to table created for the class at n-nary end (use primary key from other table)
  2) Create FK constraint, @ManyToOne(optional=false)
  3) For navigation identify the table that contains mapping, @OneToMany(mappedBy="offering")
- n-to-m Associations:
  1) Add new table representing association (2 cols)
  2) Make compound PK from 2 columns
  3) Add FK constraints to PKs of both classes
- Map Generalizations:
  1) Derive tables for all bottom-level classes
     - P: easy to get instances of specific types
     - C: hard to get supertypes
  2) Derive one table for top-level class, copy attributes from subclasses, add attribute for type of object
     - P: easy to get instances of supertype
     - C: lots of NULL values
  3) One Table per Class
     - P: No redundancy → data consistency
     - C: selecting all attributes requires JOINs
- SELECT: columns of interest
- FROM: which tables
- WHERE: query conditions
- ORDER BY: order of results (col1, col2, .. ASC|DESC)
- INNER JOIN: <table> ON <condition> (table1.col1 = table2.col2)
- LEFT JOIN: 
- RIGHT JOIN: 
- FULL OUTER JOIN: 
- JPA: Java Persistence API
- TDD:
  - tests written first
  - should shape the code so it is easily testable
- Software Frameworks: Implementations of common flows of logic with gaps that can be filled in with application code

- Framework Principles:
  1) Traditional: app manages flow of execution
  2) Inversion: framework manages
  3) Dependency Injection: dependencies in the framework code are satisfied by app code that injects those dependencies
- Spring: Relies on Dependency Injection
- Spring IoC Container: Framework-controlled engine that manages the flow of execution
- Bean: Class instantiated, assembled & managed by the IoC Container
- Business Service: Implements business functionality + calls persistence layer
- REST Controller: Access to service at specific access point (REST API), filters business data to DTOs
- Validation: "Are we building the right product?"
- Verification: "Are we building the product right?"
- Static V&V: Code Reviews
  - Moderator, Scribe, Reviewer, Reader, Producer (author)
  - Pre-Review, Review Meeting, Post-Review
  - Modern: Author, Reviewer, Integrator, Verifier



- Stub: partial implementations of components on which a CUT depends
- Driver: partial implementations that exercise and depend on the CUT
- Black/White Box:
  - Unit Tests → WB
  - System Tests → BB
  - Stress Tests → BB
  - Integration Tests → WB
  - Acceptance Tests → BB
- Missing functionality cannot be revealed by WB techniques
- Unexpected functionality cannot be revealed by BB techniques

- Statement Coverage: $\frac{\text{Executed Statements}}{\text{All Statements}}$
- Branch Coverage: $\frac{\text{Execution Branches}}{\text{All Branches}}$
  (true & false branches of all conditions)
- Path Coverage: $\frac{\text{Executed Paths}}{\text{All Paths}}$
- Unit Testing: Focuses on testing the functionality of an object or method
  - Setup, Call MUT, Assert, Teardown
- Integration Testing: Focuses on the interface that is exposed to others + combine units
- Top-Down Integration: high to low-level units
  - Few drivers, many stubs
- Bottom-Up Integration: All opposite
- System Testing: tests end-to-end scenarios
- State Transition: Source State, Target State, Trigger, Guard Condition, Action