Distributed software is an information system where requesters (clients) and service providers are separated and communicate over a network, cooperating through layers such as **presentation**, **application logic**, and **resource/data management**. Web application categories include **conventional 3-tier client-server**, **Service-Oriented Architectures (SOA)**, and **cloud computing**. A typical **3-tier web application** uses server scripts for logic and data management modules for persistence; testing approaches are distinct per layer: **UI/presentation testing** validates layout/navigation (e.g., Selenium); **server-side logic** uses unit frameworks (e.g., phpUnit) for business rules; and **database testing** employs tools like **DBUnit** to verify inserts/updates.

3-tier client-server	Presentation, application logic, and data/resource layers split across requester and provider tiers.	Let's you target tests per tier and validate contracts between layers as part of gray-box plans.
Service-oriented (SOA)	Application behavior composed from services with networked boundaries and interoperability concerns.	Encourages contract, schema, and integration tests in addition to UI flows to catch cross-service faults .
Cloud architectures	Hosted services/platforms offering elastic, distributed deployment of app tiers.	Testing must account for environment variability and scaling, requiring automation and environment-aware assertions.

Gray-box testing is emphasized as the most appropriate technique for web applications because it blends black-box end-user outcomes with white-box internal knowledge of architecture, design, and environment. It explicitly considers user-visible behavior alongside system-specific technical details like session management, data flow, and interoperability between distributed components. This approach is superior for web apps because they are loosely coupled and distributed; data for a single feature is often scattered across multiple devices and databases, meaning pure black-box tests might miss integration faults that internal knowledge can isolate. Furthermore, the heterogeneity of browsers and devices, combined with dynamic content adaptation, requires tests that target specific architecture risks and cross-component contracts rather than just surface-level outputs.

Challenges in Testing: loose coupling (failures can be network-dependent), **data scattering** (data for one feature resides on multiple DBs), **heterogeneity** (must test across Chrome/Firefox/Safari and Mobile/Desktop), and **dynamic adaptation** (content changes based on location).
Quality dimensions: Beyond functionality, web apps must be tested for **Usability** (layout/nav), **Performance** (response time under load), **Reliability** (network tolerance), **Security** (Injection/XSS), and **Interoperability** (APIs).Web app design directly constrains test design; for example, a clear separation of layers allows for focused testing of UI, server code, and databases, while the navigation structure (page flows, forms) serves as the basis for deriving user-scenario tests. For **Service-Oriented Architectures (SOA)**, testing must validate the **workflow control logic** (sequence, branching, concurrency). **Petri nets** are preferred over Decision Tables or Finite State Machines (FSMs) because they uniquely capture **concurrency** (parallel activities) and **deadlocks**. Petri nets model workflows using places (states), transitions (events), tokens (active instances), and arcs, allowing testers to systematically derive sequences that cover selection, repetition, and parallel execution paths . **Core Constructs: Places** (circles = states/conditions), **Transitions** (bars = events/activities), **Tokens** (markers = active instances), **Arcs** (flow) . **Firing Rule:** A transition is **enabled** (can fire) *only* when **ALL** its input places have at least one token. When it fires, it consumes tokens from inputs and produces tokens in outputs.

Security testing is a critical dimension for web applications, focusing on vulnerabilities such: **SQL Injection:** User input is naively concatenated into queries (e.g., `WHERE` clauses). **Prevention:** Use **Prepared Statements** / Parameterized Queries to bind input as data, not code. **Cross-Site Scripting (XSS):** Malicious scripts injected via web forms are stored and executed in *other* users' browsers (stealing cookies/sessions). Testing requires **input partitions** with malicious payloads (e.g., script tags).

Selenium is the primary toolkit for automating web application testing, consisting of Selenium IDE for recording/playback and WebDriver for robust, programmatic automation. **Selenium IDE** is useful for quick smoke tests but is limited by a lack of programming logic (loops/conditionals) and poor handling of dynamic content. It uses assertions (which abort

execution on failure) and verifications (which verify conditions but allow the script to continue) to check element presence, titles, and values. For scalable testing, **Selenium WebDriver** allows tests in languages like Java to locate elements using strategies like ID, name, or XPath and perform operations such as `driver.get()` to load pages or `sendKeys()` to input text. To address the challenges of heterogeneity and large-scale testing resources, **Selenium Grid** can be used to execute tests in parallel across different browsers and environments. **Selenium Remote Web Drivers** allows offloading the execution of test cases to a remote server or cloud infrastructure rather than burning local CPU. Use when **large-scale application** needing considerable processing power but **insufficient in-house resources.**

Code Review Checklist: **Data declaration**: correct types/lengths/storage classes and consistent initialization where provided (e.g., string vs char array). **Computation**: mixed types or sizes in arithmetic, potential overflow/underflow, and numerical stability concerns. **Control flow**: loop termination, switch defaults/breaks, unreachable or dead code indicators. **Comparison**: off-by-one or operator mistakes and careful floating-point comparisons using an EPSILON pattern when needed. **Complexity and risk**: use cyclomatic complexity bands (1–10 low risk, 11–20 moderate, 21–50 high, >50 very high/near untestable) to flag modules needing simplification or added tests.

Code smells to spot quickly: **Duplicated code** risks inconsistent fixes and latent defects when one copy changes without others being updated. **Long method** is harder to understand, review, test, and reuse, often hiding multiple responsibilities that should be split. **Large class** accumulates too many responsibilities, reducing cohesion and making changes risky and hard to localize. **Long parameter list** hurts readability and often signals missing objects or configuration encapsulation. **Temporary field** indicates attributes used only in special cases that should be localized or moved to narrow contexts.

Refactoring strategy (how to act on smells):

- Reduce duplication by extracting common behavior into a single method/class and routing all callers to it to ensure one change point and consistent behavior.
- Shorten long methods with Extract Method and clarify intent, enabling naming to communicate behavior and making logic more testable and reviewable.
- Break large classes into cohesive ones based on responsibilities and data affinity to raise cohesion and reduce the scope of changes per class.
- Replace long parameter lists with value objects or builder/config objects to improve readability and reduce call-site errors in argument order/types.
- Localize temporary fields by moving them into helper methods or small helper classes so attributes reflect true object invariants and typical usage.

OO Design: Improve design quality (not just code) using abstraction, encapsulation, inheritance, polymorphism, and quality metrics. ***Abstraction*****:** Focus on essential features; ignore irrelevant details. **Data abstraction:** defines structure (attributes). *Procedural abstraction:* defines operations (methods). **Information Hiding**: Hide internal details (algorithms, data structures, resource policies). Interact only through controlled interfaces (public methods). ***Benefits****:* Reduces side effects. Localizes design changes. Promotes encapsulation. Avoids global data dependencies. Increases software robustness and maintainability. *Public Methods(+), Private attributes(-), Protected Data Members(#).* **Inheritance**: Mechanism for code reuse and hierarchy creation. Subclasses inherit attributes and behaviors from parent classes. Supports extension and specialization of existing components. **Polymorphism**: Means “many forms.” A polymorphic reference can refer to different object types at runtime. The method executed depends on the actual object type, not the reference type. Achieved via inheritance or interfaces. Benefits: Leads to flexible, elegant, and robust software. Simplifies code as in same method behaves differently for different objects.

Modularity is the process of decomposing a software system into separate, interchangeable components that work together through well-defined interfaces. Each module should perform a specific, cohesive function and interact with other modules only when necessary. The goal of modularity is **functional independence**, meaning each module has a single, well-defined purpose (high cohesion) and minimal reliance on others (low coupling).

Coupling is the degree to which a module is “connected” to other modules in the system. It measures how strongly one component is connected to another. Ideally, software systems should aim for *low coupling*, where modules interact through simple, well-defined data interfaces rather than relying on each other’s internal workings. *High coupling*, on the other hand, means modules are heavily dependent on each other, which increases the likelihood of system-wide failures when changes are made to one part of the code. There are several types of coupling, ranked from worst to best:

Content Coupling: One module is directly connected to the inner working of another module. Meaning where one module directly accesses another's internal logic or data. Eliminate this by typically hiding the data from outside world (make it private), and then we develop a public method so that it can modify the data member as needed (hiding the implementation logic). Fix by making internals private and exposing narrow methods so callers never touch inside parts.
Common Coupling: Two modules share a global data item. Eliminate this by passing the object to separate object types, and they keep track of their own instances. Fix by deleting globals and flowing data through parameters or dedicated services, with each class owning its own state.
Control Coupling: Data in one module is used to determine the order of execution of another module. Occurs when one module controls another's behavior by passing control flags or signals. Eliminate this by making independent new methods for each scenario instead of passing parameters such as signal/control flag. Fix by splitting into explicit methods or using polymorphism/strategy so the callee decides without a flag.
Stamp Coupling: One module passes non-global data structures/objects to another module. fix by passing just the needed fields or a smaller DTO/interface.
Data Coupling: One module passes elementary data types, such as int, float, and char, as parameters to the other; this is the most desirable form. Keep parameters minimal and cohesive to avoid drifting into stamp/control coupling.

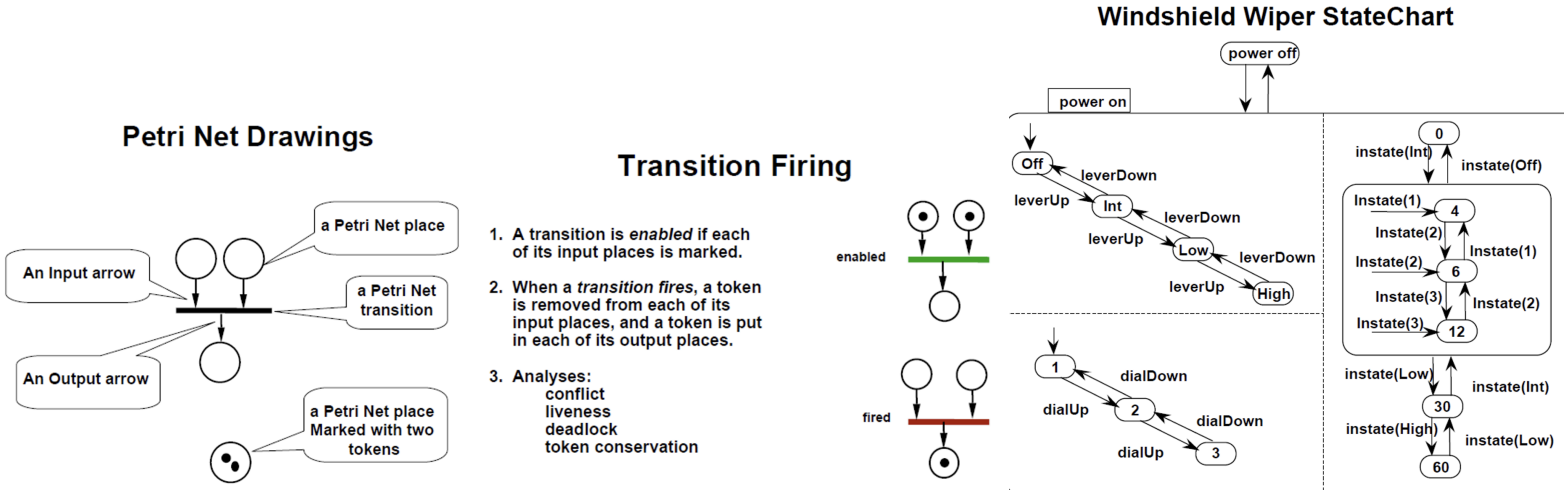
Cohesion: The degree to which all elements of a component are directed towards a single task and all elements directed towards that task are contained in a single component. In a well-designed module, all its functions and data work together to achieve a single, clearly defined purpose — this is known as **high cohesion**. Conversely, **low cohesion** occurs when a module performs multiple unrelated tasks, which makes the code confusing, difficult to maintain, and error-prone. **Fixing** it by: Splitting unrelated tasks into separate functions/classes so each unit does one meaningful job. Keep operations that work on the same data structure together, and move unrelated operations to the owners of their data. If the output of one step feeds the next, keep the pipeline together but split each step into its own small function within that pipeline module. Ensure every element contributes to one well-defined function, which is the ideal end state after refactoring.
A system with **high cohesion** and **low coupling** is considered well-designed because each component focuses on a single task and interacts minimally with others. Such systems are easier to test, maintain, and extend, leading to higher software quality. Conversely, **high coupling** and **low cohesion** result in fragile systems where a small change can cause cascading errors across multiple modules. This combination makes testing and debugging difficult, reduces reusability, and increases maintenance costs.

***C-K Metrics for OO Design*:** Chidamber and Kemerer (C-K) metrics suite 6 desirable attributes:
Weighted Methods per Class (WMC) measures the total number of methods in a class, weighted by their cyclomatic complexity. A higher WMC indicates a more complex class that is harder to test and maintain. Keeping WMC low helps maintain simplicity. **Depth of Inheritance Tree (DIT)** indicates the number of levels of inheritance from the root class. As DIT increases, a class inherits more behavior, making it harder to predict. A DIT value greater than 5 is generally considered problematic. **Number of Children (NOC)** counts the direct subclasses of a class. While inheritance promotes reuse, a very high NOC (greater than ten) may suggest excessive subclassing or misuse of inheritance, which complicates maintenance. **Response for a Class (RFC)** measures the number of methods that can be executed in response to a message received by an object. A high RFC indicates a complex class with higher coupling and testing difficulty. Typically, RFC values above 50 are discouraged. **Coupling Between Object Classes (CBO)** measures how many other classes a particular class is connected to. High CBO reduces modularity and increases the chance that changes in one class will affect others. **Lack of Cohesion in Methods (LCOM)** assesses how closely related the methods in a class are to each other. A high LCOM value means that methods do not share common data, indicating poor cohesion and a potential need for class restructuring.

OO Design Types:
Entity Classes: Long-lived business data/rules (e.g., Student, Account). *Test Focus:* Invariants, constraints, setters.
Boundary Classes: Interface with outside world (UI, APIs). *Test Focus:* Input validation, output mapping.
Control Classes: Coordinator logic/workflows. *Test Focus:* Order of execution, decision paths.

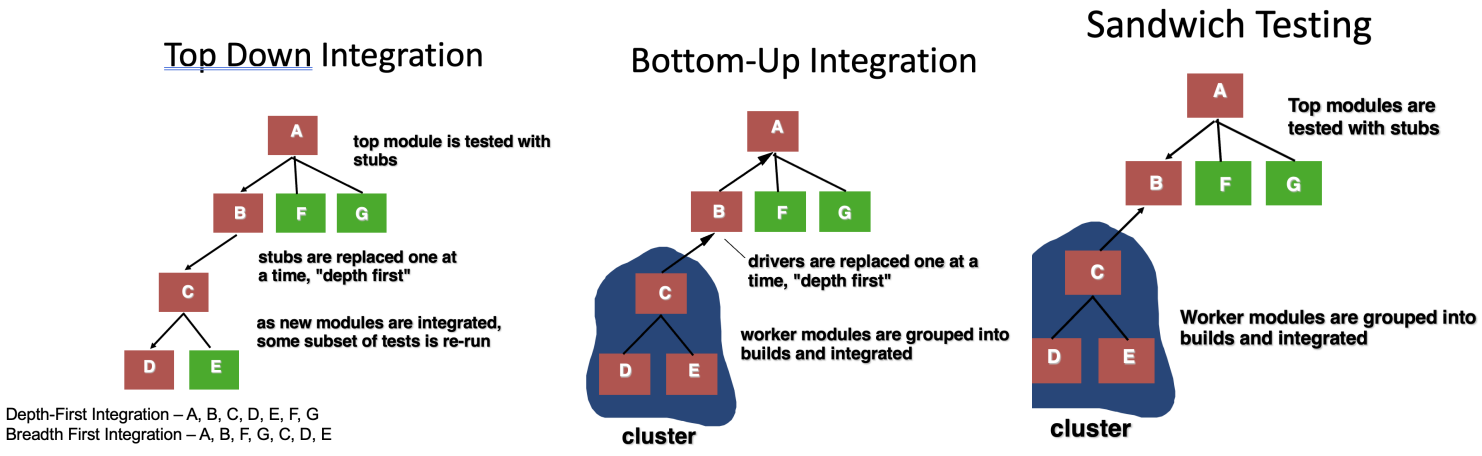
StateCharts & Petri-Nets: StateCharts and Petri-nets are critical for testing dynamic behavior where an object or system moves between states based on events. **StateChart Components**: Models states, events, transitions, guard conditions, and actions. *Deriving Test Cases:* *State Coverage:* Ensure every state is reached at least once. *Transition Coverage:* Ensure every transition is traversed at least once. *Path/Scenario Coverage:* Test critical multi-step paths, including normal flows and exceptional paths (e.g., illegal events). **Testing Procedure**: Invoke the public method that triggers an event, then verify the object ends in the expected state and the transition action occurred. Also, test robustness by triggering

illegal events in specific states. **Petri-Nets** are used to model workflows (like Web Services). *Transitions:* A transition is "enabled" if all its input places have a token. A Petri Net is a bipartite directed graph, consisting of two sets of nodes (places and transitions) and edges showing places that are either inputs to, or outputs from, transitions. Formally, a Petri Net is a four-tuple (P, T, In, Out) where P is a non-empty set of places, T is a non-empty set of transitions, In is a mapping from P to T, i.e, (a subset of P×T), and Out is a mapping from T to P (a subset of T×P). Places → circle, Transitions → line, and tokens → dot.



Designing Tests from Use Cases & Diagrams:
Use Cases: Identify preconditions, main success scenarios, alternative flows, and postconditions. *Test Design:* Treat every scenario (main + extensions) as a test case. Setup preconditions -> drive inputs -> verify postconditions match the description. *Negative Testing:* Create tests that violate preconditions to ensure robustness.
Sequence Diagrams: Describe dynamic object behavior and time-ordered messages.*Test Design:* Instantiate all participating objects (or mocks). Send messages mirroring the diagram's lifeline interactions including loops/branches. *Verification:* Check that messages result in expected state changes and the output matches the use case.
Collaboration (Communication) Diagrams: Emphasize object links and message flow. *Test Design:* Ensure every link/collaboration is exercised. Check object multiplicities (e.g., one controller vs. many entities). *Mocks:* Use these diagrams to identify where Mock Objects are needed (e.g., for external services).

Integration Testing Strategies:
Big-Bang: It builds and links all components at once and then tests their combined behavior; it simplifies build management but makes it hard to isolate faults when tests fail and is rarely recommended for complex systems.
Top-Down: Starts from the main control module and works down. *Tools:* Uses **Stubs** to replace unimplemented lower-level modules. *Pros/Cons:* Early testing of high-level control and interface; good for prototypes. However, low-level utilities are tested late, and writing stubs is time-consuming.
Bottom-Up: Starts with low-level "worker" modules and works up. *Tools:* Uses **Drivers** to simulate callers. *Pros/Cons:* Thorough testing of critical low-level utilities/services early. However, high-level logic and full system behavior are tested late.
Sandwich: Combines both. Integrates from the top and bottom simultaneously toward a "middle" layer. *Tools:* Uses both **Stubs** and **Drivers**. *Pros:* Balances early control testing with early utility testing.



QA spans three complementary strategies: Defect prevention includes education/training, domain and process knowledge, appropriate tooling, and sound architecture/design practices to reduce error injection. Defect reduction includes inspections/reviews and testing to observe failures, diagnose faults, fix them, and re-verify corrections before release. Defect containment includes software fault tolerance (e.g., recovery blocks, redundancy) to limit failure impact when faults escape earlier stages. **Specifications** (Design/Requirements) are often the major contributor to software bugs, not just coding errors. Software Testing is a defect **detection/reduction** technique, NOT a prevention technique.

Functional Testing (Blackbox testing): Specification-derived tests focusing on required behaviors. Requirements, SRS, and user-facing behaviors. Validating observable outputs for inputs and scenarios. Equivalence classes, boundary value analysis, decision tables, model-based tests. Catches specification conformance issues and user-visible failures. May miss internal faults if outputs look correct. Acceptance/system testing and user-oriented validation. Blackbox testing is synonymous with functional testing, relying on external specifications to select inputs and check outputs without internal knowledge. Code is not required.
Structural testing (Whitebox testing): Code/structure-derived tests focusing on internal elements and paths. Control flow, data flow, code structure, and coverage goals. Exercising statements, branches, paths, and data interactions. Code coverage, path testing, slicing, data-flow testing. Reveals untested code, dead code, and logic faults not visible externally. May miss missing or ambiguous requirements even with full coverage. Unit/integration testing and structural adequacy measurement. Whitebox testing is synonymous with structural testing, using internal knowledge to design tests that meet coverage criteria over code and data structures. Code is required.
Venn Diagram (SPT) Representative regions: $S \cap P \cap T$ (implemented as specified and tested), $P \cap T - S$ (extra functionality tested), $S \cap P - T$ (known missing tests), $S \cap T - P$ (tests for specified but missing features), $S - (P \cup T)$ and $P - (S \cup T)$ (likely unknown gaps), and $T - (S \cup P)$ (tested unspecified behaviors, often quality-related).
Exploratory testing is a black box approach where testers learn the system, form goals, design on-the-fly experiments, and adapt based on findings while documenting insights. Heuristics such as SFDIPOT guide exploration across Structure, Function, Data, Interfaces, Platform, Operations, and Time to uncover risk areas efficiently. Function categorization, stakeholder understanding, and coverage of unspecified but expected behaviors complement scripted functional testing.

Equivalent classes: Equivalent Partitioning subdivides the input domain into disjoint subsets where any one representative test from a class is assumed to reveal the same fault as any other in that class, which keeps tests focused and non-redundant. Steps to identify EPs: Consider both valid and invalid inputs, look for equivalent output events, look for equivalent operating environments, and organize them in a table. **EP classification: Weak normal**: choose one representative from each valid class. **Strong normal**: cartesian product over valid classes across variables. **Weak robust**: introduce a single invalid value while others are valid. **Strong robust**: cartesian product including invalid classes, assuming multiple simultaneous faults. **Limitation**: EP does not specifically target off-by-one and boundary faults. EP alone does not model inter-variable dependencies well.

Weak normal	V_a, V_b: # of **Valid** partitions for Variable A and B. I_a, I_b: # of **Invalid** partitions for Variable A and B. Formula: **$\text{Max}(V_a, V_b)$**
Strong normal	Formula: **$V_a * V_b$**
Weak robust	Formula: **Weak Normal + $I_a + I_b$**
Strong robust	Formula: **$(V_a + I_a) * (V_b + I_b)$**

Boundary value analysis: BVA holds all but one variable at nominal values and drives the remaining variable through its boundary points to reveal edge-sensitive faults. It has two modes: **Normal (General)** tests only valid boundaries using 5 points (min, min+1, nominal, max-1, max), while **Robust** adds invalid edges to test error handling using 7 points (adds min-1, max+1). It works best when inputs are independent, bounded physical quantities. Reals and chars: adapt to representable granularity. Strings: test empty, max length, and max+1 length. **Limitation**: BVA ignores logical dependencies and can produce gaps if applied naively.
Decision table: DT formalizes logical combinations by mapping conditions to actions, ensuring each (feasible) rule is considered and enabling algebraic minimization of redundant cases. Originating from **cause–effect graphing**, DTs are recommended for logically complex situations with interacting conditions. **Limitation:** Ordering of inputs, durations, and timing of events are critical, or the number of conditions explodes, requiring careful reduction.
Model-based testing: Finite State Machines represent states as nodes and transitions as edges labeled by events/data/time, with paths from start to final state forming test cases. Determinism is required for this technique; otherwise, the same input may not produce a predictable output to assert against. **Limitation**: Requires determinism

and modeling effort; pure FSMs don't cover concurrency without other models. **Use Cases**: Stateful workflows, UI flows, protocols, and device interactions. **Petri Nets:** Extension of FSMs used specifically to test **concurrency** and synchronization issues. **State Transition Diagram:** Best for capturing system behavior and valid transitions between states.
Effective Combination: A practical "pendulum" strategy balances effort and effectiveness: start with EP to remove redundancy, add BVA to probe edges, and then DT to capture cross-variable dependencies.

Bug Management: A bug report must contain enough information for teams to understand, reproduce, diagnose, and fix the problem, emphasizing clarity and credibility in reporting. Core fields typically include title, description, status, version no., feature area, reproduction steps, assignment, severity, customer impact, environment, and resolution.
Bug Workflow:
1. Report the bug
2. Confirm the bug - Bug Verification
3. Programmer evaluates the bug and fixes it, or not to fix it due to various reasons such as bug is not reproducible, not a credible bug…etc. The process of deciding which bugs to fix and which to leave in the product is called bug triage.
4. The programmer decides not to fix it and sends it to the project manager.
5. If testers and programmers disagree whether a bug should be fixed: Send it to a project team (representatives of the key stakeholder groups) reviews bugs. Such a team is called a Triage team.
6. If the bug is fixed, the programmers mark the bug fixed in the bug tracking system.
7. The test group retests the fixed bugs and close or reopen.
One of the most commonly used quantitative approaches is to measure the **"Defect Arrival Rate"**. The **Weibull curve** is used to model defect arrival behavior that typically rises and then falls over time; many practitioners expect bug-count curves to exhibit this general shape during testing.

JUnit annotations: In JUnit 4, **@Before** runs before every test method to set up a fresh test environment, such as creating new objects, resetting fields, or loading small fixtures; its JUnit 5 equivalent is *@BeforeEach*. **@After** runs after every test to clean up by closing files, deleting temporary data, or resetting external state, with *@AfterEach* as its JUnit 5 equivalent. **@BeforeClass** runs once before any test methods in the class to perform expensive or shared setup tasks like starting an embedded database or initializing caches; it must be static in JUnit 4, and its JUnit 5 equivalent is *@BeforeAll* (static unless using a per-class test instance). **@AfterClass** runs once after all tests complete to tear down shared resources such as servers or global connections; it must also be static, and its JUnit 5 equivalent is *@AfterAll*. Use *@Before*/*@After* for per-test setup and cleanup (temporary files, resetting state) and *@BeforeClass*/*@AfterClass* for shared, resource-heavy setup and teardown. Avoid common mistakes such as forgetting to make @BeforeClass/@AfterClass methods static in JUnit 4, performing heavy setup in @Before when it could go in @BeforeClass, sharing mutable state across tests without resetting it, and mixing JUnit 4 and JUnit 5 annotations.
Parameterized testing in JUnit allows a single test method to run multiple times with different input values, reusing the same assertion logic for many cases instead of writing separate tests. In JUnit 5, this is done using ***@ParameterizedTest*** together with a *source annotation* that automatically supplies test inputs. It helps eliminate duplicate test methods by separating test logic from varying data and makes it easy to cover *boundary values* and *equivalence classes* thoroughly. The core annotations include ***@ParameterizedTest*** (marks the test for repeated runs), ***@ValueSource*** (provides a list of literal values such as numbers or strings), ***@CsvSource*** (provides multiple arguments per test, e.g., input–expected pairs), **@MethodSource** (links to a factory method returning a Stream or Iterable of complex arguments), and ***@CsvFileSource*** (reads arguments from an external CSV file on the classpath). Examples include checking even numbers using @ValueSource, verifying string uppercasing with input–expected pairs via @CsvSource, and testing addition using data from a @MethodSource. Parameterized tests are ideal for *testing utility functions, validators, and small logic-heavy methods*, especially where the behavior stays constant but inputs and expected outputs vary.

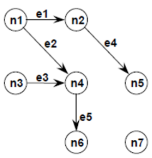
Code-Based Testing: It focus more on the statements, logic paths, and data in a program under test and test cases are directly based on testing the correctness of them. Code Based Testing Strategies: Basic path testing (Control flow coverage), Statement coverage, Decision-to-Decision testing, Dataflow coverage, Condition testing, Code based risk assessments (complexity of the code, object oriented matrices …etc).
Control-flow graph: A CFG models a program as a directed graph where nodes represent statements or statement fragments and edges represent possible transfers of control between them. **Cyclomatic Complexity:** Edges - Nodes + 2. (Make sure that there is only 1 sinking node.) OR Number of Predicate Nodes + 1.
Data Flow Testing (D-U Testing): **Definition (D):** Code that assigns a value to a variable (e.g., `int x = 0;` or `x = y + z`).

Use (U): Code that reads a variable: **P-Use (Predicate):** Used in a conditional check (e.g., if (x > 0)). **C-Use (Computation):** Used in a calculation or output (e.g., y = x + 1 or print(x)). **D-U Pair:** A pair (d, u) such that d is a defining node, u is a using node, and there is a "def-clear path" from d to u (the variable is not redefined in between).

Indegrees and Outdegrees

10. The indegree of a node in a directed graph is the number of distinct edges that have the node as an endpoint. We write indeg(n)

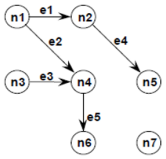
11. The outdegree of a node in a directed graph is the number of distinct edges that have the node as an start point. We write outdeg(n)



indeg(n1) = 0 outdeg(n1) = 2
indeg(n2) = 1 outdeg(n2) = 1
indeg(n3) = 0 outdeg(n3) = 1
indeg(n4) = 2 outdeg(n4) = 1
indeg(n5) = 1 outdeg(n5) = 0
indeg(n6) = 1 outdeg(n6) = 0
indeg(n7) = 0 outdeg(n7) = 0

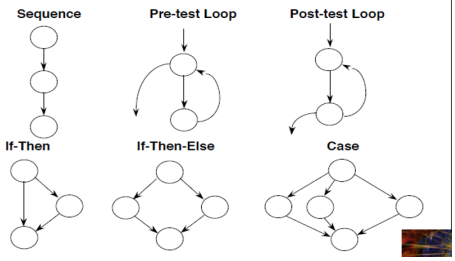
Types of Nodes

12. A node with indegree = 0 is a source node.
13. A node with outdegree = 0 is a sink node.
14. A node with indegree ≠ 0 and outdegree ≠ 0 is a transfer node. (AKA an interior node)



n1, n3 and n7 are source nodes,
n5, n6 and n7 are sink nodes,
n2 and n4 are transfer nodes.

Program Graphs of Structured Programming Constructs



Review: *Static testing* inspects artifacts (requirements, specs, designs, code, guides, test cases) without executing code, complementing dynamic testing rather than replacing it. IEEE recognizes management reviews, technical reviews, inspections, walkthroughs, and audits, each focused on fitness to purpose, standards conformance, and early defect detection across lifecycle artifacts.

Management review: management-led evaluation of plans, schedules, progress, and effectiveness of approaches to ensure fitness for purpose. **Technical review**: peer evaluation of a product's suitability and standard/specification conformance to find discrepancies before later stages. **Inspection**: formal, preparation-heavy peer examination of requirements/design/code by non-authors to detect faults and standard violations, with materials distributed beforehand. **Walkthrough**: formal author-led presentation of the artifact to a small group where reviewers listen and question to understand and spot issues. **Audit**: independent examination of product/process for compliance with specs, standards, contracts, or other criteria.

Informal Review: Desk checks, casual team meetings, or pair programming reviews. Aim: quick error discovery and continuous quality improvement. ***Pair programming*** encourages continuous review as a work product (design or code) is created. The benefit is immediate discovery of errors and better work product quality as a consequence.

Formal technical reviews (FTRs): *Objectives* include uncovering errors in function/logic/implementation, verifying requirements satisfaction, standards conformance, development uniformity, and project manageability. *Typical structure*: 3–5 participants, ≤2 hours prep each, ≤2 hours meeting, focused on a concrete work product (e.g., a design slice or a component's code). Roles: producer (author), review leader (readiness and distribution), reviewers (prepare notes), recorder (captures issues) for structured and accountable outcomes. ***Producer***—the individual who has developed the work product –informs the project leader that the work product is complete and that a review is required. ***Review leader***—evaluates the product for readiness, generates copies of product materials, and distributes them to two or three *reviewers* for advance preparation. ***Reviewer(s)***—expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work. ***Recorder***—reviewer who records (in writing) all important issues raised during the review. ***Conducting Review*:** Review the product, not the producer; set and maintain an agenda; limit debate and rebuttal; note problems without solving all in-meeting; keep written notes; require prep; use tailored checklists; allocate time/resources; train reviewers; and review the review process itself.

i.) In software quality assurance we say errors become faults and faults become failures. In here, errors refer to errors in the code..... (T/F). F

ii.) If the control-flow testing shows 100% code coverage, there is no need to functional testing (black-box) as all the code has already been tested in glass-box testing .....(T/F). F

iii.) According to C-K matrix , a good OO design has more depth in inheritance hierarchy the as it encapsulates functionality at different levels better… (T/F). F

iv.) Code inspection is more formal than walkthrough as the inspection is carried out by software professionals outside of the software development team… (T/F). T

v.) Bug triage is the process of changing a new bug to verified status.....(T/F). F

i.) Which of the following is least applicable in black box testing?

A) Specification of the system
B) Understand the environment in which the software will be deployed
C) Source code
D) Use cases

ii.) Which of the following "quality dimensions" are applicable for WebApps:
A) usability, performance, interoperability
B) security, maintainability, navigability
C) content, structure, function
D) all of the above

iii.) An objective of content testing is:
A) to uncover syntactic errors (e.g., typos, grammar mistakes) in text-based documents, graphical representations, and other media
B) to uncover semantic errors (i.e., errors in the accuracy or completeness of information) in any content object presented as navigation occurs
C) to find errors in the organization or structure of content that is presented to the end-user
D) all of the above E) none of the above

iv.) The decision table technique should be used in a situation where

A) Variables are physical variables B) Inputs are not independent
B) Variables are logical variables D) code is available to the tester

v.) Which of the following construct that cannot be modeled by the finite state model
A) Sequence B) selection C) concurrency D) repetition

vi.) Which of the following is NOT a factor that software testers use to design equivalent class-based test cases

A) valid inputs B) invalid inputs C) output D) System configuration
E) Cyclomatic complexity

vii.) Equivalent partitioning is commonly used in black-box testing.
Which of the following may not be considered in determining equivalent class?

B) Valid Input data B) Invalid input data
C) Minimum system requirement to run the software
D) Outputs D) Use case diagram

viii.) Which of the following JUnit tag is used in initializing object instances before running each and every test case?

A) @BeforeClass B) @AfterClass C) @Before D) @After

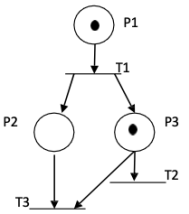
ix.) Which of the following statements best describes the functionality of the following JUnit test

```
@Test
public void Test2() {
int d=31, m=1, y=2010;

Date date = new Date(m, d, y);
date.increment();
assertEquals("test", "2/1/2010", date.toString());
}
```

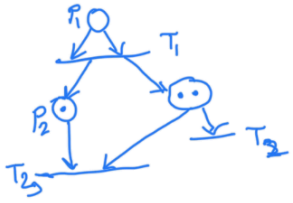
A) It checks if the date.toString() method call returns the string "test"
B) It checks of the date.toString() method returns the "2/1/2010"
C) It checks if the increment() method of the object.
D) None of the above

b.) [8 Points] Answer the following questions based on the Petri-net given below



a) What are the transitions that are enabled? T_1, T_2

b) Draw the snapshot of the Petri-Net after transition T1 is fired?



4. **Data Path Testing :** Draw the control flow graph for the following SequentialSearch method. Then design test cases to test the SequentialSearch algorithm implementation.

```
int SequentialSearch(int[] a, int searchItem)
{
int itemIndex = -1; Boolean found= false;
for(location = 0 ; location < a.length ; location++)
    {
      if(a[location] = = searchItem)
        {
          found = true;
          itemIndex = location;
          break;
        }
    }
    return itemIndex;
}
```

a) Draw Dataflow graph and then identify D-U pairs and D-U paths for following variables in the above code segment.

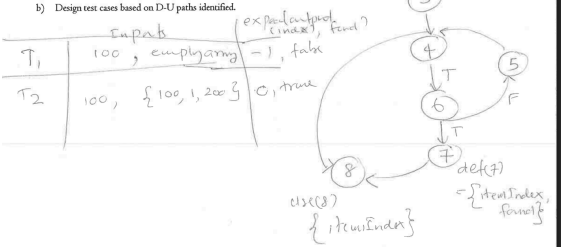
itemIndex, *found*

D-u path
itemIndex ⇒ 2→3→4→8 ⇒ 1→2→3→4→8
⇒ 7→8 ⇒ 1→2→3→4→6→7→8

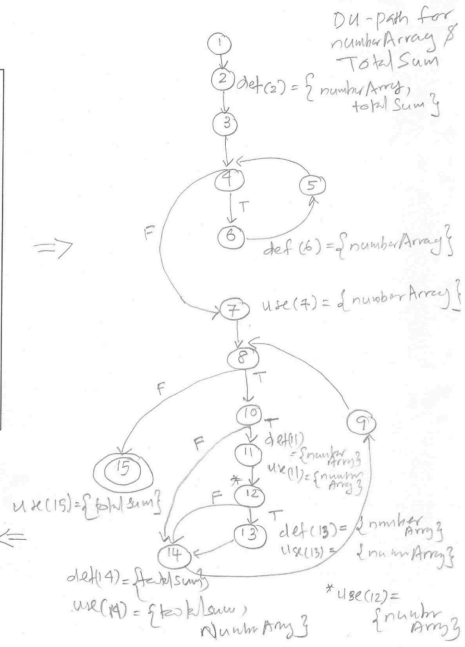
	Du Pair
~~location~~ itemIndex	(2,8) (7,8)
found	(2) (7)

b) Design test cases based on D-U paths identified.

	Inputs	expected output (index, found)
T_1	100, emptyarray	-1, false
T_2	100, {100, 1, 200}	0, true




```
public static boolean IsValidMod10Number(String number) — ①
{
        int [] numberArray = new int[number.length()];
        boolean checkBit = false;
        int sumTotal =0;
                for(int i=0; i < number.length() ;i++)
                {
                        numberArray[i] = (int) number.charAt(i);
                }
                for(int index = numberArray.length -1 ; index >= 0 ; index--)
                {
                        if(checkBit)
                        {
                        numberArray[index] *=2;
                                if(numberArray[index] > 9)
                                {
                                numberArray[index] -= 9;
                                }
                        }
                sumTotal += numberArray[index];
                checkBit = !checkBit;
                }
        return sumTotal % 10 ==0;
}
```



Du-Pairs

Variable	Du-pair
totalSum	(2,14) (2,15) (14,14) (14,15)
numberArray	(2,-), (6,7), (6,11) (11,12), (11,13), ([6,11,13], 14) (or)

StateChart Based Testing

- StateChart is a techniques that provides visual representation of states of objects, transitions, and change of states through transitions.
- StateCharts can be used where the current action of an objects depends on previous actions.
- State chart is a collection of "blobs" that represent state and arrows that represent transitions.
- A blob can contain another blob
- Also StateCharts can be used to represent concurrent states of more than one object

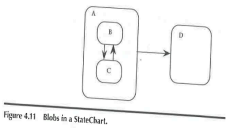


Figure 4.11 Blobs in a StateChart.

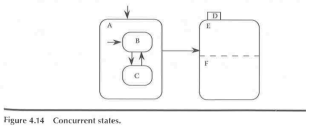


Figure 4.14 Concurrent states.

Du-paths

totalSum:

pair(2,14) P1: 2→3→4→7→8→15

pair(2,15) P2(1): 2→3→4→[6→5→4]*→7→8→[10→11→14→9→8]*→15

P2(2): 2→3→4→[6→5→4]*→7→8→[10→11→12→13→14→9→8]*→15

P2(3):

pair(14,14): No path

pair(14,15) P2(3): 14→15

P3 is part of P2

Compromised paths

P1: Empty string

P2(1): String with one digit

P2(2): String with at least two digits and the second digit >4

P2(3): String with atleast two digits and all digits are less than < 5

Test cases

Test case #	Test data (input)	expected output (totalSum)	
1 (P1)	number = ""	0	
2 (P2.1)	number = "6"	6	
3 (P2.2)	number = "1321"	10	← (1+4+3+2)
4 (P2.3)	number = "1567"	17	← (7+3+5+2)