Solutions to Exercise Problems

Software Testing: Techniques, Principles, and Practices, by JJ Shen

Chapter 2 – Software Test Cases (Exercise Problems)

P2.1 — "Which of the following are test cases?" (classification + convert to real tests)

(1) Random sampling of 50 balls repeated 100 times; average the 100 numbers and verify the average is between 22 and 28.

A. Is it a test case?
No (as written).

B. Why not?

- It relies on **randomness**: a single run does not produce a single deterministic expected value — outcome is a sample from a probability distribution.

- It is an **aggregate/statistical check**, not an atomic input→expected-output test with a single pass/fail oracle.
- Reproducibility requires controlling or recording random seeds and the sampling method; independence is weak because the experiment bundles 100 trials into one check.

C. How to make it a proper test:
You have two correct options depending on intent:

1. **Statistical test plan (keep randomness):** specify sampling procedure, sample size, acceptance criteria and significance level. Example:
    - Population: 100 balls (50 red, 50 blue). Action: for i=1..100 draw 50 without replacement, record Xi = number of red balls.
    - Compute mean X^{-}\bar XX¯ and sample standard deviation s. Use a two-sided test (t or z) at α = 0.01. Accept if $X^{-} \in [22,28]$\bar X \in [22,28]$X^{-} \in [22,28]$ **and** p-value > 0.01 (or 95% CI contained in [22,28]). Log RNG seeds, environment, and number of trials for reproducibility.
2. **Deterministic unit test (remove randomness):** mock or seed the RNG so the draws are reproducible. Example: with seed S, the routine produces exactly these 100 counts → assert computed average equals the expected deterministic number (or equals a specific value). This tests aggregation logic, not the true randomness.

Either way, document the pass criteria, sampling assumptions, and how to reproduce runs.

(2) Verify that the number of stars in the Milky Way Galaxy is 300,200,190,515.

A. Is it a test case?
No (as written).

B. Why not?

- This is a **factual/numeric claim about the real world**, not a software input→output behavior you can deterministically verify in a unit/test harness.
- The quantity is essentially **unobservable with exactness** and subject to scientific estimation and change; it lacks a software oracle and is not reproducible in a test environment.
- The statement has no clear *testable* procedure in the software sense — no defined inputs, no deterministic method that produces that exact number.

C. How to make it testable / useful:

- If the goal is to test a **software function** that returns a published estimation value (e.g., `getMilkyWayStarEstimate()`), then make the spec explicit:
    - **Deterministic check:** If the function is supposed to return a stored constant, assert equality to the documented constant.
    - **Tolerance/Source check:** If the function computes/ingests literature estimates, assert the returned number **matches a trusted source** or falls within a specified tolerance or range and include the source/version (e.g., “value must

equal the published estimate from NASA 2024 within ±5%").

- If the intent is a scientific measurement, treat it as **a measurement/estimation process**, not a unit test. Provide measurement method, uncertainties, and acceptance intervals.

(3) Open the calculator app on your mobile device, verify 2 plus 2 is 12.

A. Is it a test case?
Yes — technically (but the *expected result* is wrong for correct behavior).

B. Why (yes):

- The statement identifies a concrete action (enter 2 + 2) and a single expected result (12). That matches the chapter's *structural* definition of a test case: atomic, deterministic, reproducible, and with a single oracle.
- **Important caveat:** a test case can assert an incorrect expectation — it will simply fail. The test is syntactically valid, but semantically wrong if the *correct* result is 4.

C. Notes / better version:

- If your intent is to verify the calculator **is correct**, the correct expected result is 4. So the proper test case should be:
  - **TC:** Open calculator, enter 2 + 2, expected = 4.
- If the goal is to **detect a known bug** that wrongly returns 12, then the test as written is a regression test asserting

the buggy behavior; but for correctness you should assert 4 and mark an observed result of 12 as a failure (bug).
- Additional helpful variants: test integer vs floating mode, locale decimal separators, leading/trailing whitespace, long chains of operations, and idempotency after clearing.

(4) Go to google.com, enter search term: "orange fruit", verify that the 3rd link in the result is www.myorangefruitsite.com.

A. Is it a test case?
No — not reliably (as written).

B. Why not?

- Search results are **non-deterministic** and highly dependent on environment factors: personalization, cookies, location, user history, time, search engine ranking changes, A/B experiments, and even the exact user agent.
- Because of that variability, the outcome (what appears as the 3rd link) is **not reproducible** across machines or runs unless the environment is strictly controlled. That violates the chapter's determinism and reproducibility requirements for a proper atomic test case.

C. How to make it a proper test (deterministic or manageable):

1. **Control the environment** — if you must test search results:

    - Use a fixed environment (clean browser profile, no personalization, fixed geo/IP or use a specific Google Search API endpoint with fixed parameters), fixed time if possible, and a stable index snapshot (if available). Document these controls in the test.
    - Use the non-personalized SERP API if available, and assert the 3rd result equals the expected URL.
    - Even then, this will be brittle and likely to break as rankings change; treat it as a *monitoring/check* rather than a unit test.
2. **Convert to a robust requirement:** instead of exact rank, assert **presence** of www.myorangefruitsite.com in the top N (e.g., top 10) under controlled, repeatable query conditions — or verify the site appears for a fixed cached snapshot of the results.
3. **Alternative:** If the software under test is *your own search service*, then you can make the ranking deterministic for a test index and assert the 3rd link. In that case, create and use a **test index** with deterministic documents and ranking rules.

Short summary table

Item	Is test case?	Reason (chapter rules)	How to fix / convert
(1) Random sampling average	**No**	Statistical, non-deterministic aggregate	Formal statistical test plan (α, CI) or mock RNG for

Item	Is test case?	Reason (chapter rules)	How to fix / convert
			deterministic unit test
(2) Milky Way star count	**No**	Real-world estimate, not a reproducible software oracle	If software returns a stored constant, assert equality; otherwise use tolerance/source-based check
(3) 2 + 2 = 12	**Yes (syntactically)**	Atomic & deterministic — but expected value is incorrect for correct behavior	Correct test should expect 4; as written it will fail (unless testing a known bug)
(4) Google 3rd link = site	**No (not reliable)**	Search rankings vary with environment → not reproducible	Control environment or use API/snapshot; assert presence in top-N instead of exact rank

P2.2: Write test cases for the following software module.

A door is controlled by one push button, if the door is open, push the button will close it, if the door is closed, push the

button will open it, if the door is jammed, push the button will have no effect. Write test cases for this application. Discuss if the tests are finite, are encapsulating, are genuine, are deterministic, are reproducible, are atomic, and are independent.

Solutions:

Problem restated:
We have a door controlled by a single push button:

- If the door is **open**, pushing the button → **close**.
- If the door is **closed**, pushing the button → **open**.
- If the door is **jammed**, pushing the button → **no effect**.

We need to:

1. Write test cases.
2. Discuss the nature of the tests: are they **finite, encapsulating, genuine, deterministic, reproducible, atomic, independent**?

Part 1: Test cases

We can model the door as a finite-state machine with three states: **Open, Closed, Jammed**.

Test Case Set

TC1 – Closed → Open

- Precondition: Door = Closed

- Action: Press button
- Expected: Door transitions to Open

TC2 – Open → Closed

- Precondition: Door = Open
- Action: Press button
- Expected: Door transitions to Closed

TC3 – Jammed → (No change)

- Precondition: Door = Jammed
- Action: Press button
- Expected: Door remains Jammed (no effect)

TC4 – Sequence Open → Closed → Open

- Precondition: Door = Open
- Actions: Press → becomes Closed; Press again → becomes Open
- Expected: Correct toggling behavior

TC5 – Sequence Closed → Open → Closed

- Precondition: Door = Closed
- Actions: Press → becomes Open; Press again → becomes Closed
- Expected: Correct toggling behavior

TC6 – Jammed transition stability

- Precondition: Door = Jammed
- Actions: Press button repeatedly
- Expected: Always stays Jammed

Part 2: Discussion of test properties

- **Finite** ✅
  - Yes. The system has only **three states** (Open, Closed, Jammed) and one input (button press). That makes the test space finite — 3 input-state combinations cover the system.
- **Encapsulating** ✅
  - Yes. These test cases fully cover all *possible input/state combinations*. No valid behavior is left untested.
- **Genuine** ✅
  - Yes. Tests match real user actions and observable outcomes (pressing the button, seeing door open/close/unchanged).
- **Deterministic** ✅
  - Yes. For a given state and button press, the next state is uniquely determined by the rules.
- **Reproducible** ✅
  - Yes. The same precondition (door state) and same action (press button) will always yield the same result.
- **Atomic** ✅
  - Each test case describes one **action + single expected outcome**. (Even TC4/TC5 can be split into two atomic tests if strict atomicity is desired.)
- **Independent** ✅
  - Yes, if we set up the door state before each test (reset to Open/Closed/Jammed as needed). No test relies on another's result.

Final Answer (Concise Form)

Test cases:

1. Closed + press → Open
2. Open + press → Closed
3. Jammed + press → Jammed
4. Open + press + press → Closed → Open
5. Closed + press + press → Open → Closed
6. Jammed + repeated press → always Jammed

Discussion:

- Tests are **finite** (3 states × 1 action).
- They are **encapsulating** (cover all behaviors).
- They are **genuine** (realistic user interactions).
- They are **deterministic** (each state-action pair has one defined result).
- They are **reproducible** (same input = same output).
- They are **atomic** (one action → one expected result, though sequences can be split).
- They are **independent** (resetting preconditions makes tests self-contained).

P2.3: *Black Holes.

Black holes are the densest, most massive singular objects in the universe. The no-hair theorem states that, once it achieves a stable condition after formation, a black hole has only three independent physical properties: mass, charge, and angular momentum. An event horizon is a boundary in space-time beyond which events cannot affect an outside observer. At the event horizon, the escape velocity is equal to the speed of light. Since general relativity states that nothing

can travel faster than the speed of light, nothing inside the event horizon can ever cross the boundary and escape beyond it, including light. Each black hole has its own event horizon.
Consider black hole as a software application, tests are needed to verify that a stable black hole with certain properties (mass, charge, and angular momentum) has a certain event horizon. In other words, given a set of values of mass, charge, and angular momentum, the size of the event horizon isCHAPTER 2: SOFTWARE TEST CASES 27
defined. Is this system testable in this regard? If testable, in what sense the test cases are finite? are deterministic? are encapsulate? are reproducible?

Solution:

Problem restated:
We treat a black hole as if it were a "software application." Inputs are the three physical parameters — **mass, charge, angular momentum**. Output is the **size of the event horizon** (radius). The question asks: Is this system testable? If yes, are test cases finite, deterministic, encapsulating, reproducible?

1. Is this system testable?

Yes — in the **software testing sense**.

- The event horizon size is mathematically **defined by formulas from general relativity**.
- For a given set of inputs (mass, charge, angular momentum), there is a unique, computable result.
- Even though we cannot observe a real black hole's event horizon directly, the *software program implementing the formula* is **testable against the mathematical specification**.

So: **Testable as a software model**, not as a physical experiment.

2. Are the test cases finite?

- **No**, strictly speaking.
    - Inputs (mass, charge, angular momentum) are real numbers → infinite domain.
- **Yes**, in practice.
    - Testers can select a *finite subset* of representative values using techniques like **equivalence partitioning** and **boundary value analysis** (e.g., small/large mass, zero vs extreme charge, max/min spin).

3. Are the test cases deterministic?

- **Yes.**
    - The event horizon formula is deterministic: same input values always produce the same result.
    - Floating-point precision or numerical approximations may introduce tiny differences, but conceptually the mapping is deterministic.

4. Are the test cases encapsulating?

- **Partially.**
    - We can cover main equivalence classes and boundary conditions (e.g., uncharged Schwarzschild black hole, extremal charged or rotating black holes).
    - But because the input space is infinite, the test set cannot *fully* encapsulate every possible input.
    - However, within the chosen partitions, tests *do* encapsulate the expected behavior.

5. Are the test cases reproducible?

- **Yes.**
    - Given the same input values (mass, charge, angular momentum) and the same algorithm/precision environment, the output will always be the same.
    - As long as implementation details (floating-point libraries, units) are controlled, tests are reproducible.

Final Answer (Concise Form)

- **Is it testable?** → Yes, as a software system (not physically).
- **Finite?** → No (inputs are continuous), but in practice we can design a finite *representative* set.
- **Deterministic?** → Yes, the mathematical formulas produce unique results.
- **Encapsulating?** → Not fully (infinite space), but representative partitions can encapsulate expected classes.
- **Reproducible?** → Yes, same input = same output under controlled environment.

P2.4: A web application has three input boxes for the day, month, and year, and produces the number of days since January 1, 1970. List 5

input vectors and 5 output vectors for this application.

Input Vectors

Each input vector is a tuple: (day, month, year)

1. (1, 1, 1970) – The epoch date itself.
2. (2, 1, 1970) – The day after the epoch.
3. (1, 1, 2000) – Y2K date.
4. (15, 8, 1947) – A date **before 1970** (to test negative day counts).
5. (31, 12, 2025) – A future date.

Output Vectors

Each output vector is the **number of days since January 1, 1970**. I'll calculate them carefully **step by step**.

1. **Input:** (1, 1, 1970)
    - Days since Jan 1, 1970 = **0**
2. **Input:** (2, 1, 1970)
    - One day after epoch = **1**
3. **Input:** (1, 1, 2000)
    - Count leap years between 1970–1999.
        - Leap years: 1972, 1976, 1980, 1984, 1988, 1992, 1996 → 7 leap years
        - Total years = 30 → 30 × 365 + 7 = 10,957 days
    - Days since Jan 1, 1970 = **10,957**
4. **Input:** (15, 8, 1947)
    - Before epoch → negative count.
    - Years from 1947 to 1969 = 23 years

- Leap years in 1948, 1952, 1956, 1960, 1964, 1968 → 6 leap years
- Days = 23 × 365 + 6 = 8,401
- Days from Jan 1, 1947 to Aug 15, 1947 = Jan→Jul = 31+28+31+30+31+30+31=212 + 15 (Aug) = 227
- Total days **before epoch** = 8,401 + 227 = 8,628 → Negative: **-8,628**

5. **Input:** (31, 12, 2025)
   - Years 1970–2024 = 55 years
   - Leap years: 1972, 1976, 1980, 1984, 1988, 1992, 1996, 2000, 2004, 2008, 2012, 2016, 2020, 2024 → 14 leap years
   - Days = 55 × 365 + 14 = 20,089
   - Days in 2025 up to Dec 31 = 365 (2025 not leap) → Total = 20,089 + 365 = **20,454**

Summary Table

Input (day, month, year)	Output (days since Jan 1, 1970)
(1, 1, 1970)	0
(2, 1, 1970)	1
(1, 1, 2000)	10,957
(15, 8, 1947)	-8,628
(31, 12, 2025)	20,454

P2.5: Classify the following tests into fundamental types.

(1) Verify that after a superuser, 'User_A', is signed in on a membership website, 'User_A' can see the address information of another user: 'User_B'.

- This checks **functional correctness** of a feature.
- **Type:**
  - Functional testing → 1
  - System testing → 2 (it involves full system behavior)
  - Security testing → 2 (access control check)
  - Black box testing → 1–2 (testing from user perspective)
  - UI testing → 1–2 (observing displayed information)

(2) Verify after turning a web page with English (the default language for the site) into a different language, such as French, the form on the page displays properly.

- This checks **UI display, compatibility, and internationalization**.
- **Type:**
  - System testing → 2
  - Internationalization testing → 2
  - UI testing → 1–2
  - Usability testing → 2
  - Compatibility testing → 2
  - Browser testing → 2 (if testing across browsers)

(3) Verify under a fixed set of settings of environment variables, such as on the same browser and network connection, a specific web page loads between 1.5 and 2.0 seconds.

- This is **performance-focused**.
- **Type:**
  - Performance testing → 2
  - Benchmark testing → 2.0
  - Load testing → 2.1 (if multiple concurrent users are implied)
  - Stress testing → 2.2 (if extreme conditions are tested)
  - System testing → 2

(4) A user logged on to a shopping website, verify the user can click on the shopping cart link and see the product items in the shopping cart.

- Checks **functional behavior of a core feature**.
- **Type:**
  - Functional testing → 1
  - System testing → 2
  - Black box testing → 1–2
  - UI testing → 1–2
  - Acceptance testing → 3 (if end-user perspective is considered)

(5) A Java method, average(int a, int b), is written to calculate the average of two numbers, test:

```
@Test
public void testAverageOfTwoIntegersThreeAndFive () { ... }
```

- This is **low-level code verification**.
- **Type:**
  - Unit testing → 0
  - White box testing → 0–2
  - Backend testing → 0–2 (method logic in backend)
  - Regression testing → 0–2 (if used in automated test suites)

Summary Table (Regenerated with Your Scheme)

Test #	Description	Test Types
1	Superuser sees another user's address	Functional (1), System (2), Security (2), Black box (1–2), UI (1–2)
2	Page displays properly in French	System (2), Internationalization (2), UI (1–2), Usability (2), Compatibility (2), Browser (2)
3	Page loads in 1.5–2.0 seconds	Performance (2), Benchmark (2.0), Load (2.1), Stress (2.2), System (2)
4	User sees shopping cart items	Functional (1), System (2), Black box (1–2), UI (1–2), Acceptance (3)
5	Java method `average()` unit test	Unit (0), White box (0–2), Backend (0–2), Regression (0–2)

Chapter 3 – Equivalence Partitioning (Exercise Problems)

P3.1 Price Variable

Spec: Value must be **> 0** and **< 1,000,000**; if **< 10,000** ⇒ **A (reject)**; if **≥ 10,000** ⇒ **B (accept)**. Generate base, base-worst, and standard cases.

- **Partitions**
  - **Invalid:** $x \leq 0$; $x \geq 1{,}000{,}000$
  - **Valid/A:** $0 < x < 10{,}000$ → action A
  - **Valid/B:** $10{,}000 \leq x < 1{,}000{,}000$ → action B
- **Base case (one per class, interior values):**
  $x = -1$ (invalid), $x = 5{,}000$ (A), $x = 50{,}000$ (B), $x = 1{,}000{,}000$ (invalid)
- **Base worst case (add a second rep where risk is higher):**
  Add $x = 0$ (invalid boundary), $x = 9{,}999$ (A high edge), $x = 10{,}000$ (B low edge), $x = 999{,}999$ (B high edge).
- **Standard case (add boundary neighbors):**
  { 0, 0.01, 9,999, 10,000, 10,001, 999,999, 1,000,000, 1,000,001 }.
  (If the system only accepts integers, use 1, 9,999, 10,000, 10,001, 999,999 and keep 0, 1,000,000 as invalid boundaries.)

P3.2 BMI Calculation

Valid ranges: weight W ∈ [1,200] kg; height H ∈ [1,300] cm (convert to meters before BMI=W/H²). Build base + standard with boundaries.

- **Partitions:**
  W invalid (<1, >200), W valid [1..200]; H invalid (<1, >300), H valid [1..300].
- **Base case:** choose interiors + a couple invalids
  - Valid interior: (W,H) = (70, 175) → BMI ≈ 70 / 1.75² ≈ 22.86
  - Invalid W low: (0, 170); Invalid W high: (201, 170)
  - Invalid H low: (70, 0); Invalid H high: (70, 301)
- **Standard case (boundaries ±1):**
  - W: 1, 2, 199, 200
  - H: 1, 2, 299, 300
  - Sample pairing to keep atomic:
    (1,300), (2,299), (199,2), (200,1) — plus invalid neighbors: (0,170), (201,170), (70,0), (70,301).
  - Compute expected BMI numerically for valid pairs (rounding rules per product spec).

P3.3 Go Board Coordinates

Board coordinates are valid in [1..19]×[1..19]. Find partitions and list standard boundary tests.

- **Partitions:**
  x or y < 1 (invalid); 1 ≤ x,y ≤ 19 (valid); x or y > 19 (invalid).

- **Standard boundary tests (representative):**
  Valid corners: (1,1), (1,19), (19,1), (19,19)
  Edge neighbors: (2,1), (1,2), (18,19), (19,18)
  Invalid lows: (0,5), (5,0), (0,0)
  Invalid highs: (20,5), (5,20), (20,20)
  Valid interior: (10,10)

P3.4 Safe Add Program

Given code detects overflow when both inputs have same sign but sum flips sign. Design EP + boundary tests.

- **Partitions:**
  (A) a≥0, b≥0 (non-negatives)
  (B) a<0, b<0 (negatives)
  (C) mixed signs (no overflow by same-sign rule)
- **Boundaries (32-bit int):** MIN=−2,147,483,648; MAX=2,147,483,647
  (Use product's int width; here assume Java int.)
- **Tests:**
  - Mixed signs: (5, −3) → 2 (no overflow path)
  - Non-neg interior: (123, 456) → 579
  - Neg interior: (−123, −456) → −579
  - **Overflow positive:** (MAX, 1) → overflow message path (sum wraps negative)
  - **Overflow negative:** (MIN, −1) → overflow message path (sum wraps non-negative)
  - **Edges that don't overflow:** (0,0)→0; (0,MAX)→MAX; (0,MIN)→MIN

P3.5 Next Date Problem

Design EP + boundary tests for computing the next calendar date.

- **Partitions (month-day families):**
  - 31-day months: Jan, Mar, May, Jul, Aug, Oct, **Dec**
  - 30-day months: Apr, Jun, Sep, Nov
  - February:
    - Common year: 28 days
    - Leap year: 29 days (year divisible by 4 and not by 100 unless also divisible by 400)
- **Key classes & tests:**
  - Middle of month: 2025-05-15 → 2025-05-16
  - 30-day month end: 2025-04-30 → 2025-05-01
  - 31-day month end (non-Dec): 2025-07-31 → 2025-08-01
  - **Year roll-over:** 2025-12-31 → 2026-01-01
  - Feb common year: 2025-02-28 → 2025-03-01
  - Feb leap year: 2024-02-28 → 2024-02-29; 2024-02-29 → 2024-03-01
  - Invalid dates (reject): 2025-04-31; 2025-02-30; 2025-13-01; 2025-00-10
  - Boundary neighbors: 2025-01-01, 2025-01-31, 2025-03-01

P3.6: A Simplified Email Field

Test an email field on a registration form, the requirements are:
(1) The format is emailId@domain.
(2) The length of 'emailId' should be between 3 and 64.
(3) Only letters (A-Z, a-z) and numbers (0-9) are accepted for 'emailId'
.

(4) The length of ‘domain’ should be between 5 and 64.
(5) One and only one dot (.) in ‘domain’ is accepted, and it cannot be right after ‘@’, nor any of the last two characters.
Generate test cases for standard case of equivalence partitioning.

Requirements Recap

1. **Format:** emailId@domain
2. **emailId length:** 3–64 characters
3. **emailId characters:** only A-Z, a-z, 0-9
4. **domain length:** 5–64 characters
5. **domain format:** exactly one dot (.), not immediately after @ or in the last two characters

Step 1: Identify equivalence classes

Email ID (emailId)

- Valid class (V1): 3–64 letters/numbers
- Invalid classes (I1–I3):
    - I1: < 3 characters
    - I2: > 64 characters
    - I3: contains invalid characters (special chars, spaces)

Domain

- Valid class (V2): 5–64 chars, exactly one dot in allowed position
- Invalid classes (I4–I8):
    - I4: < 5 characters
    - I5: > 64 characters
    - I6: no dot

    - I7: more than one dot
    - I8: dot at invalid position (right after @ or last two chars)

Overall format

- Valid format (V3): contains exactly one @ separating emailId and domain
- Invalid format (I9): missing @ or multiple @

Step 2: Standard Case Test Cases (Equivalence Partitioning)

Test Case	Input Email	Expected Result	Equivalence Class
1	abc@domain.com	Valid	V1 + V2 + V3
2	ab@domain.com	Invalid – emailId too short	I1
3	aVeryLongEmailIdExceedingSixtyFourCharacters1234567890@domain.com	Invalid – emailId too long	I2
4	abc!@domain.com	Invalid – invalid character in emailId	I3
5	abc@dom.co	Valid – domain length 5, dot in middle	V2
6	abc@dom	Invalid – domain too short	I4
7	abc@verylongdomainnamethatisdefinitelymorethan64characterslongabcdefghijk.com	Invalid – domain too long	I5
8	abc@domaincom	Invalid – no dot in domain	I6
9	abc@do.main.com	Invalid – multiple dots	I7
10	abc@.domain.com	Invalid – dot right after @	I8
11	abc@domain.c.	Invalid – dot in last two characters	I8
12	abcdomain.com	Invalid – missing @	I9
13	ab@domain@com	Invalid – multiple @	I9

Notes

- Each test case covers **one equivalence class** (standard case).
- This set can be used as a **baseline for boundary value analysis** later.
- Valid emails: satisfy all 5 rules; invalid emails: violate exactly one rule per test.

Chapter 4 – Combinatorial Testing (Exercise Problems)

P4.1 Laptop Catalogs (pairwise vectors)

Params: Brand(6)={A..F}, RAM(4)={16,32,64,128}, Disk(3)={512–1023,1024–2047,2048–4095}, OS(3)={Windows,ChromeOS,MacOS}. Build **pairwise** set.

One workable 12-case covering array (each pair across any two factors appears ≥1×):

1. A,16,512–1023,Windows
2. A,32,1024–2047,ChromeOS
3. A,64,2048–4095,MacOS
4. B,32,512–1023,MacOS
5. B,64,1024–2047,Windows
6. B,128,2048–4095,ChromeOS

7. C,64,512–1023,ChromeOS
8. C,128,1024–2047,MacOS
9. C,16,2048–4095,Windows
10. D,128,512–1023,Windows
11. D,16,1024–2047,MacOS
12. D,32,2048–4095,ChromeOS
13. E,16,512–1023,MacOS
14. E,32,1024–2047,Windows
15. E,64,2048–4095,ChromeOS
16. F,64,512–1023,Windows
17. F,128,1024–2047,ChromeOS
18. F,16,2048–4095,MacOS

Notes:

- 12 is a common lower bound for 4-factor 3/4-level coverage, but because Brand has 6 levels, a **slightly larger** set (e.g., 18) helps distribute Brand pairs without holes. Above 18 rows achieve robust pair coverage while keeping variety by cycling values, matching the book's practice of adding variety when filling empty cells.

P4.2 Five 3-way Switches (On/Off/Auto)

Construct pairwise and 3-wise with orthogonal arrays.

- **Pairwise (strength 2):** Use OA **L18($2^1 \cdot 3^7$)**; map the 5 switches S1..S5 to five of the 3-level columns and ignore the 2-level column. 18 runs cover every pair across S1..S5 at least once. (Any standard L18 layout is acceptable; label levels {0,1,2} ↔ {On, Off, Auto}.)

- **3-wise (strength 3):** Use OA **L27(3^{13})** and take any 5 columns for S1..S5. 27 runs cover every triple across S1..S5 at least once. (Again, standard L27 layout; levels map to On/Off/Auto.)

(Exact OA tables aren't reproduced here—choose any canonical L18/L27 and map columns as above; the book stresses using OAs for construction and warns about the exponential growth when increasing strength.)

Chapter 5 – Decision Table Testing (Exercise Problems)

P5.1 Movie Theater Ticket Pricing

Build decision table and derive tests.

- **Conditions (C):**
  - C1: Before 12 noon? {Y,N}
  - C2: Age band ∈ {<2, 2–12, 13–60, 60+}
- **Actions (A):** Price ∈ {0, 7, 8, 14, 16, 20, 10} (depends on C1,C2)
- **Rules (minimal, 8 total):**

Rule	C1 (Before 12)	C2 (Age)	Price
R1	Y	<2	0
R2	Y	2–12	7

Rule	C1 (Before 12)	C2 (Age)	Price
R3	Y	13–60	10
R4	Y	60+	8
R5	N	<2	0
R6	N	2–12	14
R7	N	13–60	20
R8	N	60+	16

- **Test set (one per rule, with boundary ages):**
  - R1: 11:59, age 1 → 0
  - R2: 11:59, age 2 → 7; plus boundary neighbor age 12 → 7
  - R3: 11:59, age 13 →10; neighbor 60 →10
  - R4: 11:59, age 61 → 8
  - R5: 12:00, age 1 → 0
  - R6: 12:00, age 2 →14; neighbor 12 →14
  - R7: 12:00, age 13 →20; neighbor 60 →20
  - R8: 12:00, age 61 →16

(These reflect decision-table construction and boundary picks emphasized in Ch.5.)

P5.2: Student Housing Arrangement

A school provides on-campus housing for certain groups of students. Depends on a few selections, a student is directed to a specific housing unit. Here are the specifications:
(1) single male domestic students, go to unit A.
(2) single female domestic students, go to unit B.
(3) single international students, go to unit C.
(4) students with families, go to unit D.
Use decision table technique to construct a decision table and generate test

cases from it.

Step 1: Identify Conditions and Actions

Conditions (inputs that affect the housing decision):

1. Student type: Single or with family
2. Gender (for singles): Male or Female
3. Nationality (for singles): Domestic or International

Actions (housing unit assigned):

- Unit A
- Unit B
- Unit C
- Unit D

Step 2: Define Possible Condition Values

Condition	Possible Values
Single / Family	Single, Family
Gender	Male, Female (only relevant if Single)
Nationality	Domestic, International (only relevant if Single)

Step 3: Build the Decision Table

Rule #	Single/Family	Gender	Nationality	Housing Unit
1	Single	Male	Domestic	Unit A
2	Single	Female	Domestic	Unit B
3	Single	–	International	Unit C

Rule #	Single/Family	Gender	Nationality	Housing Unit
4	Family	–	–	Unit D

Notes:

- – means the condition **does not affect the decision** for that rule.
- This table covers **all valid combinations** based on the specifications.

Step 4: Optional Extended Table (All Combinations for Clarity)

Rule #	Single/Family	Gender	Nationality	Housing Unit
1	Single	Male	Domestic	Unit A
2	Single	Female	Domestic	Unit B
3	Single	Male	International	Unit C
4	Single	Female	International	Unit C
5	Family	Male/Female	Domestic/International	Unit D

This extended table makes it **explicit for all gender/nationality combos**.

P5.3: NextMonth Function

A function NextMonth() takes a date as input, and outputs next month. For example, if the input is ‘June 25, 2025’, then NextMonth() returns ‘July’; if the input is ‘Dec 31, 2025’, then the output is ‘January’. The input can be any day of any year (say, between 1999 and 2099).

Use decision table technique to construct a decision table and generate test cases from it.

Step 1: Identify Conditions and Actions

Conditions (factors affecting next month):

1. Current month = December? (Yes/No)
    - If Yes → next month is January
    - If No → next month = current month + 1
2. Current day = any day (not relevant for month output in this simplified function)
3. Current year = any year (not relevant for month output, unless for leap-year consideration, but for month name only, irrelevant)

Action:

- Output: Next month name

Step 2: Define Condition Values

Condition	Values
Is current month December?	Yes, No

Step 3: Construct Decision Table

Rule #	Current Month = December?	Output (Next Month)
1	No	Current month + 1
2	Yes	January

- For **No**, we cycle to the next month in order (January → February → … → November → December).

- For **Yes** (December), wrap around to January.

Step 4: Generate Test Cases from the Decision Table

Test Case #	Input Date	Current Month = December?	Expected Output
1	June 25, 2025	No	July
2	November 15, 2025	No	December
3	December 1, 2025	Yes	January
4	December 31, 2025	Yes	January
5	January 10, 2025	No	February

Notes

- This decision table **only considers the month wrap-around**.
- The day of the month and year are irrelevant for this simplified function.
- The test cases cover:
  - Normal month → next month
  - December → wrap-around to January

P5.4: Use extended decision table technique to generate test cases for Example 5.4: Loan Rate under Different Conditions.

Step 1: Identify Causes and Effects

Causes (conditions)

Cause #	Condition
1	R ≤ 20%
2	20% < R ≤ 40%
3	R > 40%
4	S < 600
5	600 ≤ S < 700
6	700 ≤ S < 750
7	S ≥ 750

Effects (outcomes)

Effect	Loan decision
a	Rate 5.0%
b	Rate 4.5%
c	Rate 4.0%
d	Reject application

Step 2: Construct Extended Decision Table (Boolean Conditions)

Rule #	1: R ≤ 20%	2: 20%<R≤40%	3: R>40%	4: S<600	5: 600≤S<700	6: 700≤S<750	7: S≥750	a:5%	b:4.5%	c:4%	d:Reject
1	0	1	0	0	1	0	0	1	0	0	0
2	0	1	0	0	0	1	0	0	1	0	0

Rule #	1: R ≤ 20%	2: 20%<R≤40%	3: R>40%	4: S<600	5: 600≤S<700	6: 700≤S<750	7: S≥750	a:5%	b:4.5%	c:4%	d:Reject
3	0	1	0	0	0	0	1	0	0	1	0
4	1	0	0	0	0	0	0	0	0	1	0
5	0	0	1	0	0	0	0	0	0	0	1
6	0	0	0	1	0	0	0	0	0	0	1

A 1 indicates the condition/effect is true; a 0 indicates false.
This table is based on the **cause-effect graph** mapping from the specification.

Step 3: Generate Test Cases

Test Case #	R Value	S Value	Expected Effect
1	25%	650	5.0% (Rule 1)
2	25%	720	4.5% (Rule 2)
3	25%	780	4.0% (Rule 3)
4	15%	700	4.0% (Rule 4)
5	45%	650	Reject (Rule 5)
6	30%	580	Reject (Rule 6)

Notes

- The **extended decision table technique** helps reduce the number of test cases compared to standard equivalence partitioning.
- Each test case **covers one rule** from the decision table.
- Invalid zones (like R<0, S<300, or R>100%) are **not included** here; they can be added separately if testing robustness.

- This method ensures **all logical combinations of conditions that lead to unique effects** are tested efficiently.

Visual Decision Table: Loan Rate Example

Causes (conditions) are on the left, **Effects (results)** on the right. Arrows indicate which causes lead to which effects.

Causes Effects
---
1. R ≤ 20% ----------> c. Rate 4.0%
2. 20%<R≤40% + 5. 600≤S<700 --> a. Rate 5.0%
2. 20%<R≤40% + 6. 700≤S<750 --> b. Rate 4.5%
2. 20%<R≤40% + 7. S≥750 --> c. Rate 4.0%
3. R>40% ----------> d. Reject
4. S<600 ----------> d. Reject

Explanation

1. **Cause 1**: R ≤ 20% and S ≥ 600 → c (4.0%)
2. **Cause 2**: 20% < R ≤ 40% → combine with S ranges:
    - S 600–699 → a (5.0%)
    - S 700–749 → b (4.5%)
    - S ≥ 750 → c (4.0%)
3. **Cause 3**: R > 40% → reject
4. **Cause 4**: S < 600 → reject

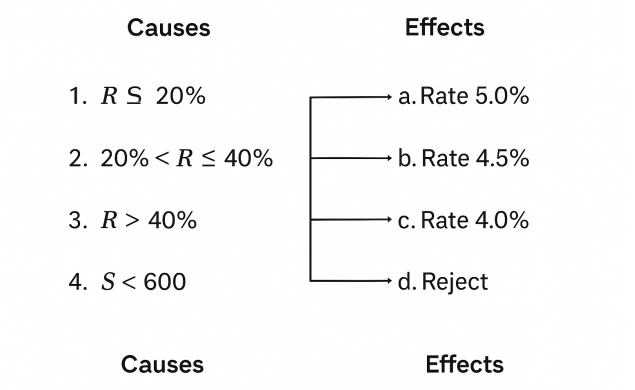
This diagram **clearly shows which conditions map to which loan rates**, making it easier to generate test cases systematically.

Optional Enhanced Table (for report)

Rule #	R Range	S Range	Effect
1	≤ 20%	≥ 600	4.0%

Rule #	R Range	S Range	Effect
2	20%–40%	600–699	5.0%
3	20%–40%	700–749	4.5%
4	20%–40%	≥ 750	4.0%
5	> 40%	any	Reject
6	any	< 600	Reject

This table is easier to include in a **formal report**, while the cause-effect diagram is great for **visual understanding**.



Chapter 6 STRUCTURAL TESTING

P6.1: A Method to Reverse a Number

The following code segment is an implementation in Java to reverse a number.

```
public static int numberReversing (int n)
{
int r = 0;
while (n != 0) {
int digit = n % 10;
r = r * 10 + digit;
n /= 10;
}
return r;
}
```

(1) Define a test graph for this method.
(2) Draw control flow graph.
(3) Identify TR(NC) and TR(EC), and construct test paths for both NC and TC.
(4) Construct simple paths and compute prime paths. identify TR(PPC).
(5) Construct test paths for PPC and write test cases for PPC

Step 1: Define a Test Graph

A **test graph** represents the flow of execution through the method.

Nodes:

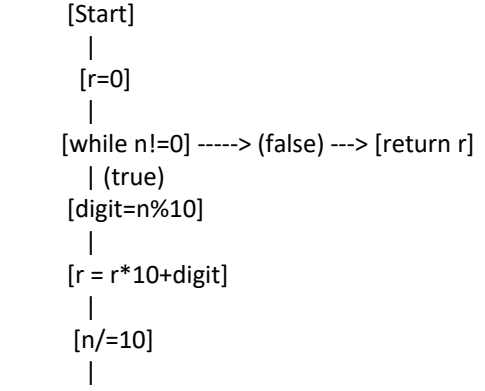
1. Start
2. r = 0
3. while (n != 0) (condition)
4. digit = n % 10
5. r = r * 10 + digit
6. n /= 10
7. Return r
8. End

Edges:

- 1 → 2
- 2 → 3
- 3 → 4 (if n != 0)
- 4 → 5
- 5 → 6
- 6 → 3 (loop back)
- 3 → 7 (if n == 0)
- 7 → 8

Step 2: Control Flow Graph (CFG)

CFG (simplified):



[while n!=0] (loop)

Step 3: Identify TR(NC) and TR(EC) and Test Paths

TR(NC) – Node Coverage

- Cover **all nodes at least once**.
- **Test paths for NC:**
    1. n = 0 → nodes: 1 → 2 → 3 → 7 → 8
    2. n > 0 → nodes: 1 → 2 → 3 → 4 → 5 → 6 → 3 → 7 → 8

TR(EC) – Edge Coverage

- Cover **all edges at least once**.
- **Test paths for EC:**
    1. n = 0 → edges: 1-2, 2-3, 3-7, 7-8
    2. n = 123 → edges: 1-2, 2-3, 3-4, 4-5, 5-6, 6-3, 3-4, 4-5, 5-6, 6-3, 3-7, 7-8

Step 4: Construct Simple Paths and Prime Paths

Simple paths: paths without repeating nodes (except possibly start/end).

- SP1: 1 → 2 → 3 → 7 → 8 (n=0)
- SP2: 3 → 4 → 5 → 6 → 3 (loop body, repeatable)

Prime paths: simple paths not a subpath of any other simple path.

- PP1: 1 → 2 → 3 → 7 → 8
- PP2: 3 → 4 → 5 → 6 → 3

TR(PPC) – Test Requirements for Prime Path Coverage:

- Each prime path must be executed at least once.

Step 5: Construct Test Paths for PPC and Test Cases

Test Path 1 (PP1):

- n = 0 → nodes: 1 → 2 → 3 → 7 → 8
- **Expected output:** 0

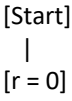
Test Path 2 (PP2):

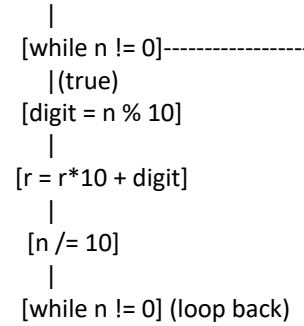
- n = 123 → nodes: 3 → 4 → 5 → 6 → 3 (executed loop at least once)
- Continue looping until n = 0: 1 → 2 → 3 → 4 → 5 → 6 → 3 → 4 → 5 → 6 → 3 → 7 → 8
- **Expected output:** 321

Summary Table of Test Cases

Test Case	Input n	Expected Output	Covers
TC1	0	0	PP1 (n=0)
TC2	123	321	PP2 (loop executed)
TC3	5	5	Loop executes once
TC4	1200	21	Loop executes, trailing zeros removed

Control Flow Graph (CFG)





Nodes Numbered

Node	Statement
1	Start
2	r = 0
3	while n != 0 (condition)
4	digit = n % 10
5	r = r*10 + digit
6	n /= 10
7	return r
8	End

Simple Paths (SP)

1. **SP1:** 1 → 2 → 3 → 7 → 8 (n = 0, loop never entered)
2. **SP2:** 3 → 4 → 5 → 6 → 3 (loop body, repeats while n != 0)

Prime Paths (PP)

- **PP1:** 1 → 2 → 3 → 7 → 8
- **PP2:** 3 → 4 → 5 → 6 → 3

Test Paths for Prime Path Coverage (PPC)

Test Case 1 (PP1):

- Input: n = 0
- Path: 1 → 2 → 3 → 7 → 8
- Output: 0

Test Case 2 (PP2):

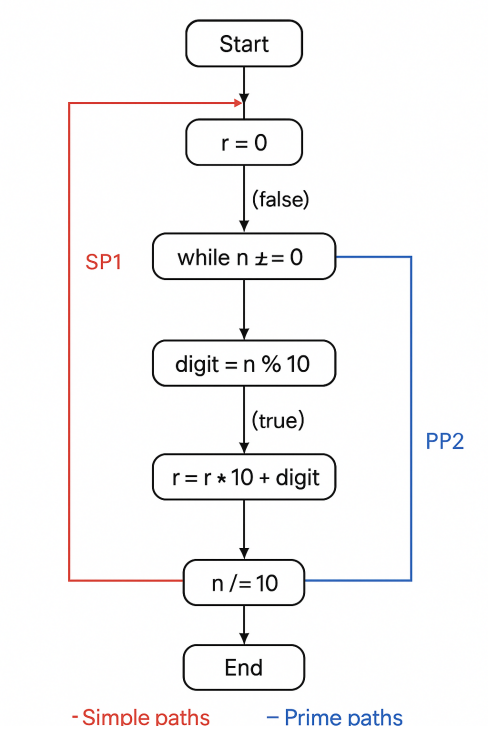
- Input: n = 123
- Path: 1 → 2 → 3 → 4 → 5 → 6 → 3 → 4 → 5 → 6 → 3 → 7 → 8
- Output: 321

Test Case 3 (Loop executes once):

- Input: n = 5
- Path: 1 → 2 → 3 → 4 → 5 → 6 → 3 → 7 → 8
- Output: 5

Test Case 4 (Loop executes multiple times with zeros):

- Input: n = 1200
- Path: 1 → 2 → 3 → 4 → 5 → 6 → 3 → 4 → 5 → 6 → 3 → 7 → 8
- Output: 21



P6.2: A Method to Determine if a Number is a Perfect Square

The code we are analyzing is:

```
public static boolean isPerfectSquare(int n) {
   int x = n % 10;
   if (x == 2 || x == 3 || x == 7 || x == 8) {
      return false;
   }
   for (int i = 0; i <= n / 2 + 1; i++) {
      if ((long) i * i == n) {
         return true;
      }
   }
   return false;
}
```

Step 1: Control Flow Graph (CFG)

The nodes and edges of the CFG can be represented as:

Nodes:

1. Start
2. `x = n % 10`
3. `x == 2 || x == 3 || x == 7 || x == 8`?
4. `return false` if condition true
5. `i = 0` initialization of loop
6. `i <= n / 2 + 1`? (loop condition)
7. `(long) i * i == n`?
8. `return true` if condition true
9. `i++` and go back to loop condition
10. `return false` after loop
11. End

Edges:

- 1 → 2

- 2 → 3
- 3 → 4 (true)
- 3 → 5 (false)
- 5 → 6
- 6 → 7 (true)
- 6 → 10 (false)
- 7 → 8 (true)
- 7 → 9 (false)
- 9 → 6
- 4 → 11
- 8 → 11
- 10 → 11

(CFG can be drawn with decision diamonds at nodes 3, 6, 7.)

Step 2: Test Requirements for Node Coverage (NC) and Edge Coverage (EC)

TR(NC) — Test paths to cover **all nodes at least once**:

- Node 1 → 2 → 3 → 4 → 11 (shortcut: x = 2, 3, 7, or 8)
- Node 1 → 2 → 3 → 5 → 6 → 10 → 11 (no perfect square)
- Node 1 → 2 → 3 → 5 → 6 → 7 → 8 → 11 (perfect square)

TR(EC) — Test paths to cover **all edges at least once**:
We need to cover both true/false for all decisions and the loop:

- Edge path for x shortcut (3 → 4 → 11)
- Loop executes at least once: 3 → 5 → 6 → 7 → 8 → 11
- Loop executes multiple times, failing to find square: 3 → 5 → 6 → 7 → 9 → 6 → 10 → 11

Step 3: Simple Paths & Prime Paths

Simple Paths: Paths that do not repeat any edge (except start/end):

1. 1 → 2 → 3 → 4 → 11
2. 1 → 2 → 3 → 5 → 6 → 10 → 11
3. 1 → 2 → 3 → 5 → 6 → 7 → 8 → 11
4. 1 → 2 → 3 → 5 → 6 → 7 → 9 → 6 → 10 → 11

Prime Paths:

- Paths that are simple and not subpaths of any other simple path.
- Prime paths here:
    1. 3 → 5 → 6 → 7 → 8 → 11
    2. 3 → 5 → 6 → 7 → 9 → 6 → 10 → 11

TR(PPC) — Test paths to cover all **prime paths**:

- Path covering prime path 1: x not in {2,3,7,8}, n is perfect square
- Path covering prime path 2: x not in {2,3,7,8}, n is not perfect square

Step 4: Test Paths for PPC & Test Cases

Test Path	Condition	Example n	Expected Result
1	x = 1, perfect square	16	true
2	x = 1, not perfect square	15	false
3 (shortcut)	x = 2, 3, 7, 8	12	false

(This ensures prime paths are executed.)

Step 5: Cyclomatic Complexity

Cyclomatic complexity formula:

$V(G)=E-N+2PV(G) = E - N + 2PV(G)=E−N+2P$

Where:

- E = number of edges = 13
- N = number of nodes = 11
- P = number of connected components = 1

$V(G)=13-11+2*1=4V(G) = 13 - 11 + 2*1 = 4V(G)=13−11+2∗1=4$

Cyclomatic complexity = 4, which **matches the number of independent paths for PPC**.

Step 6: Compound Condition Coverage (CCC)

Compound condition: (x == 2 || x == 3 || x == 7 || x == 8)

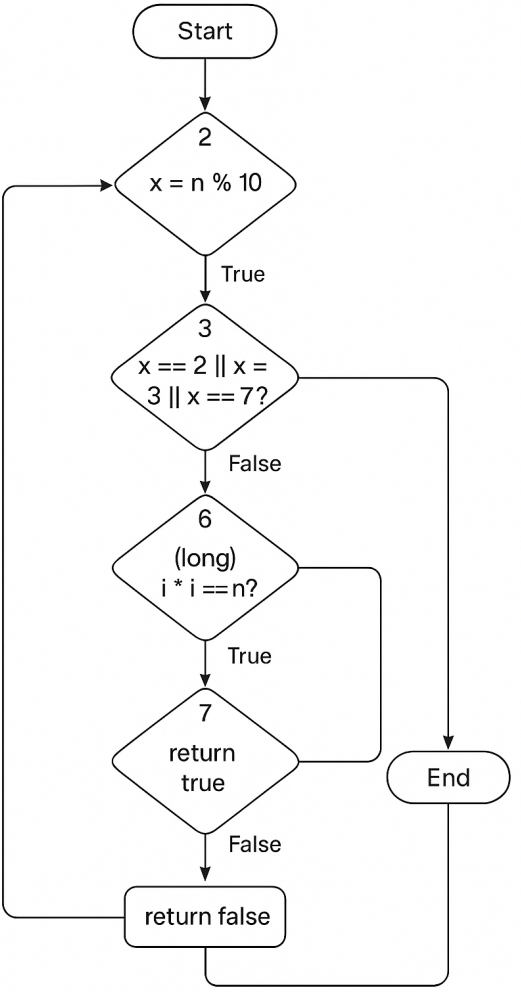
Test Cases for Compound Condition Coverage:

x	Condition evaluation
2	true (covers first OR)
3	true (covers second OR)
7	true (covers third OR)
8	true (covers fourth OR)
0	false (covers all ORs false)

Corresponding n examples: 12, 13, 17, 18, 10

Summary

- **CFG**: nodes and edges as above
- **TR(NC)**: covers all nodes
- **TR(EC)**: covers all edges
- **Prime paths**: two paths (found in loop and exit)
- **PPC test cases**: 16, 15, 12
- **Cyclomatic complexity**: 4
- **Compound condition coverage test cases**: 12, 13, 17, 18, 10


Start
2
x = n % 10
True
3
x == 2 || x =
3 || x == 7?
False
6
(long)
i * i == n?
True
7
return
true
False
return false
End

P6.3: Construct Test Paths from Prime Paths

Problem Breakdown

1. **Input:**
    - A **graph** defined by:
        - Nodes: $V = \{v1, v2, \ldots\}$
        - Initial nodes: $I \subseteq V$
        - Final nodes: $F \subseteq V$
        - Edges: $E = \{(v1, v2), \ldots\}$
2. **Output:**
    - A set of **test paths** that "tour" all **prime paths**.
3. **Definitions:**
    - **Path:** A sequence of nodes connected by edges.
    - **Simple path:** A path that does not repeat any nodes.
    - **Prime path:** A simple path that is not a subpath of any other simple path.
    - **Test path:** A path starting at an initial node and ending at a final node, which covers one or more prime paths.

Algorithm Outline

Step 1: Generate all simple paths in the graph.
Step 2: Identify all prime paths (simple paths that are not subpaths of any longer simple path).
Step 3: Construct test paths that include all prime paths. This can be done by extending each prime path to start at an initial node and end at a final node.

Python Implementation

```
from collections import defaultdict

class Graph:
    def __init__(self, nodes, edges, initial_nodes, final_nodes):
        self.nodes = nodes
        self.edges = edges
        self.adj = defaultdict(list)
        self.initial_nodes = initial_nodes
        self.final_nodes = final_nodes

        for u, v in edges:
            self.adj[u].append(v)

    def all_simple_paths(self):
        """Generate all simple paths in the graph"""
        paths = []

        def dfs(path, visited):
            last = path[-1]
            paths.append(path[:])
            for neighbor in self.adj[last]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    path.append(neighbor)
                    dfs(path, visited)
                    path.pop()
                    visited.remove(neighbor)

        for node in self.nodes:
            dfs([node], {node})
        return paths

    def prime_paths(self):
        """Compute prime paths from all simple paths"""
        all_paths = self.all_simple_paths()
        prime_paths = []

        for p in all_paths:
```

```
            is_subpath = False
            for q in all_paths:
                if p != q and self.is_subpath(p, q):
                    is_subpath = True
                    break
            if not is_subpath:
                prime_paths.append(p)
        return prime_paths

    @staticmethod
    def is_subpath(p, q):
        """Check if p is a contiguous subpath of q"""
        for i in range(len(q) - len(p) + 1):
            if q[i:i+len(p)] == p:
                return True
        return False

    def construct_test_paths(self):
        """Construct test paths that cover all prime paths"""
        prime_paths = self.prime_paths()
        test_paths = []

        for prime in prime_paths:
            # Extend prime path to initial node
            for init in self.initial_nodes:
                for final in self.final_nodes:
                    path = self.extend_path(init, prime, final)
                    if path:
                        test_paths.append(path)
        # Remove duplicates
        unique_paths = []
        for path in test_paths:
            if path not in unique_paths:
                unique_paths.append(path)
        return unique_paths

    def extend_path(self, start, prime, end):
        """Try to create a path from start -> prime -> end"""
        # Simple BFS to start
        from collections import deque
        queue = deque([[start]])
```

```
        visited = set()
        while queue:
            path = queue.popleft()
            last = path[-1]
            if last == prime[0]:
                path += prime[1:]
                break
            for neighbor in self.adj[last]:
                if neighbor not in path:
                    queue.append(path + [neighbor])
        else:
            return None  # No path from start to prime start

        # Now extend from prime end to final node
        last = path[-1]
        queue = deque([[last]])
        while queue:
            p = queue.popleft()
            last_node = p[-1]
            if last_node == end:
                path += p[1:]
                return path
            for neighbor in self.adj[last_node]:
                if neighbor not in p:
                    queue.append(p + [neighbor])
        return None

# Example usage
nodes = ['A','B','C','D']
edges = [('A','B'),('B','C'),('C','D'),('B','D'),('A','C')]
initial_nodes = ['A']
final_nodes = ['D']

graph = Graph(nodes, edges, initial_nodes, final_nodes)
prime_paths = graph.prime_paths()
test_paths = graph.construct_test_paths()

print("Prime paths:")
for p in prime_paths:
    print(p)
```

```
print("\nTest paths:")
for t in test_paths:
    print(t)
```

Explanation

1. all_simple_paths() – DFS-based search for all simple paths in the graph.
2. prime_paths() – Checks each simple path to see if it's a subpath of another; if not, it's prime.
3. construct_test_paths() – For each prime path:
    - Finds a path from an **initial node** to the start of the prime path.
    - Extends to a **final node**.
    - Collects all such test paths covering prime paths.

Example Graph

Nodes: A, B, C, D
Edges: A→B, B→C, C→D, B→D, A→C
Initial Node: A
Final Node: D

Step 1: Identify Prime Paths

1. Simple paths:

- A → B
- A → B → C
- A → B → C → D
- A → B → D
- A → C
- A → C → D
- B → C

- B → C → D
- B → D
- C → D

2. **Prime paths** (paths not subpaths of any other simple path):

- A → B → C → D
- A → B → D
- A → C → D

Step 2: Extend Prime Paths to Test Paths

- **Goal:** Test paths start at **initial node A** and end at **final node D**, covering the prime path.

Prime Path 1: A → B → C → D

- Already starts at A and ends at D → Test path: **A → B → C → D**

Prime Path 2: A → B → D

- Already starts at A and ends at D → Test path: **A → B → D**

Prime Path 3: A → C → D

- Already starts at A and ends at D → Test path: **A → C → D**

Step 3: Resulting Test Paths

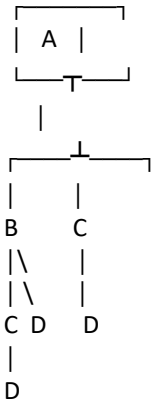
Test Paths:
1. A → B → C → D

2. A → B → D
3. A → C → D

✅ All prime paths are covered by these test paths.

Diagram

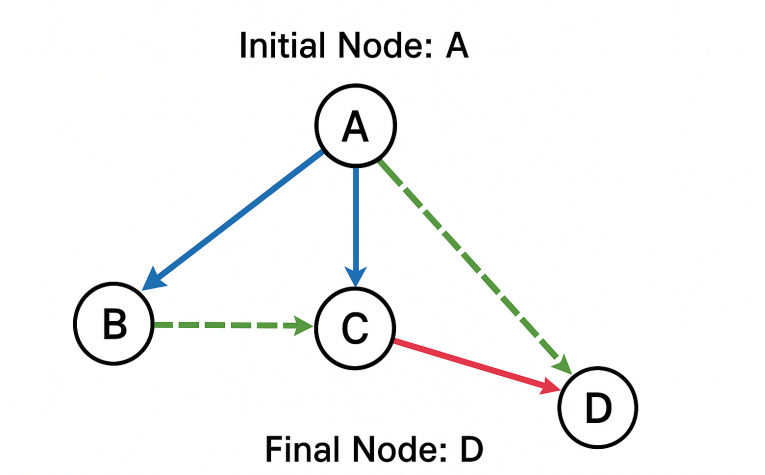
Initial Node: A
Final Node: D



Prime paths highlighted in arrows:

1. **A → B → C → D** → solid line
2. **A → B → D** → dashed line
3. **A → C → D** → dotted line

This diagram shows how all prime paths start at the initial node and end at the final node, forming test paths.



P6.4: Compound Boolean Expression: ((((a OR b) AND c) OR d) AND e)

Step 0: Understand the expression

Break it down:

1. a OR b → true if a or b is true.
2. (a OR b) AND c → true if (a OR b) is true **and** c is true.
3. ((a OR b) AND c) OR d → true if either (a OR b) AND c is true **or** d is true.
4. (((a OR b) AND c) OR d) AND e → final output true if previous result **and** e is true.

1️⃣ Basic Condition Coverage (BCC)

Definition: Each condition (a, b, c, d, e) should be **true at least once and false at least once**, **regardless of combinations**.

We can use **minimal vectors**:

a	b	c	d	e	F
0	0	0	0	0	0
1	0	0	0	1	0
0	1	1	0	1	1
1	1	1	1	1	1

Each variable is 0 and 1 at least once.

2️⃣ Condition and Edge Coverage (CEC)

Definition: Each **atomic condition** should affect the outcome **at least once in a decision**.
We need combinations where **flipping one variable changes F**.

a	b	c	d	e	F
0	0	1	0	1	0
1	0	1	0	1	1
0	1	1	0	1	1
0	0	1	1	1	1
0	0	1	0	0	0

Each variable individually affects F in at least one scenario.

3️⃣ Compound Condition Coverage (CCC)

Definition: Consider **all possible combinations of basic conditions**.
Since we have 5 variables, total combinations = 25=322^5 = 3225=32.

For brevity, we only **show all unique combinations that result in F = 1**, which ensures coverage of compound outcomes:

a	b	c	d	e	F
0	1	1	0	1	1
1	0	1	0	1	1
1	1	1	0	1	1
0	0	0	1	1	1
0	1	0	1	1	1
1	0	0	1	1	1
1	1	0	1	1	1
0	0	1	1	1	1
0	1	1	1	1	1
1	0	1	1	1	1
1	1	1	1	1	1

Covers all **true outputs** for all possible combinations of input conditions.

4️⃣ Modified Condition/Decision Coverage (MCDC)

Definition: Each **condition** should independently affect the outcome of the **decision**.

We need to pick **pairs of vectors where flipping only one variable changes F**. Example:

a	b	c	d	e	F
0	0	1	0	1	0
0	1	1	0	1	1
1	0	1	0	1	1
0	0	0	1	1	1
0	0	1	0	0	0

Each variable **independently demonstrates its effect** on the final decision.

Let's create a **complete table** for this Boolean expression F = ((((a OR b) AND c) OR d) AND e) that covers **Compound Condition Coverage (CCC)** and **Modified Condition/Decision Coverage (MCDC)**.

Since we have **5 variables**, the full truth table has 25=322^5 = 3225=32 rows. I'll also **mark which vectors are needed for MCDC**.

Step 1: Generate all 32 input combinations

#	a	b	c	d	e	F
1	0	0	0	0	0	0
2	0	0	0	0	1	0

#	a	b	c	d	e	F
3	0	0	0	1	0	0
4	0	0	0	1	1	1
5	0	0	1	0	0	0
6	0	0	1	0	1	0
7	0	0	1	1	0	0
8	0	0	1	1	1	1
9	0	1	0	0	0	0
10	0	1	0	0	1	0
11	0	1	0	1	0	0
12	0	1	0	1	1	1
13	0	1	1	0	0	0
14	0	1	1	0	1	1
15	0	1	1	1	0	0
16	0	1	1	1	1	1
17	1	0	0	0	0	0
18	1	0	0	0	1	0
19	1	0	0	1	0	0
20	1	0	0	1	1	1
21	1	0	1	0	0	0
22	1	0	1	0	1	1
23	1	0	1	1	0	0
24	1	0	1	1	1	1
25	1	1	0	0	0	0
26	1	1	0	0	1	0
27	1	1	0	1	0	0
28	1	1	0	1	1	1
29	1	1	1	0	0	0

#	a	b	c	d	e	F
30	1	1	1	0	1	1
31	1	1	1	1	0	0
32	1	1	1	1	1	1

F is computed as: F = ((((a OR b) AND c) OR d) AND e). You can check that all 32 combinations are covered.

Step 2: MCDC Vectors Selection

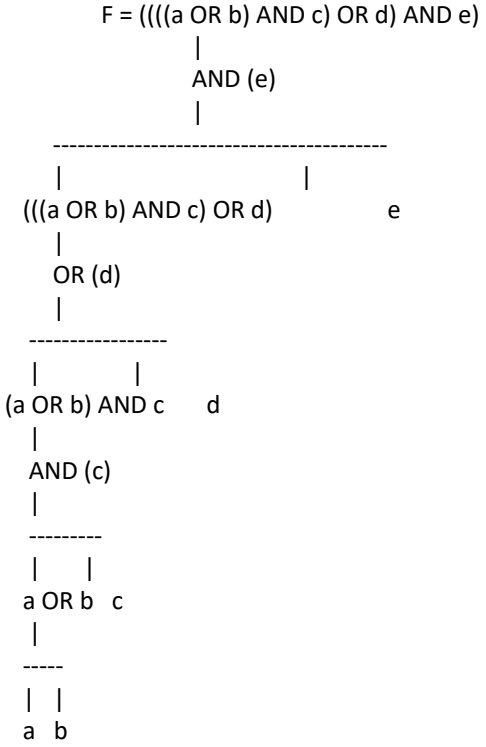
For **MCDC**, we need pairs of vectors showing **each variable independently affects F**:

Variable	Vector 1 (F=?)	Vector 2 (F=?)	Note
a	0 0 1 0 1 → F=0	1 0 1 0 1 → F=1	Flip a changes F
b	1 0 1 0 1 → F=1	0 1 1 0 1 → F=1	Flip b needed in context c=1, e=1, d=0
c	0 1 0 0 1 → F=0	0 1 1 0 1 → F=1	Flip c changes F
d	0 0 0 0 1 → F=0	0 0 0 1 1 → F=1	Flip d changes F
e	0 0 0 1 0 → F=0	0 0 0 1 1 → F=1	Flip e changes F

These **5 pairs** are enough to satisfy **MCDC**.

Let's create a **diagram/tree** showing how each variable affects the final decision F=((((a OR b) AND c) OR d) AND e)F = ((((a \text{ OR } b) \text{ AND } c) \text{ OR } d) \text{ AND } e)F=((((a OR b) AND c) OR d) AND e). This will make it clear how **MCDC** works and why each variable can independently affect F.

Step 1: Breakdown as a tree



Step 2: How each variable affects F

1. **Variable a**
   - Flips a changes (a OR b) → affects (a OR b) AND c → may affect (((a OR b) AND c) OR d) → may affect F if e=1.
2. **Variable b**
   - Same logic as a. Flipping b changes (a OR b) → affects rest of the tree.
3. **Variable c**
   - Flipping c changes (a OR b) AND c → affects (((a OR b) AND c) OR d) → affects F if e=1.

4. **Variable d**
    - Flipping d changes (((a OR b) AND c) OR d) → affects F if e=1.
5. **Variable e**
    - Flipping e changes the final AND → directly affects F regardless of previous result.

Step 3: Visualizing MCDC Pairs on the Tree

Variable: a
  0 → 1 changes (a OR b) → ... → F

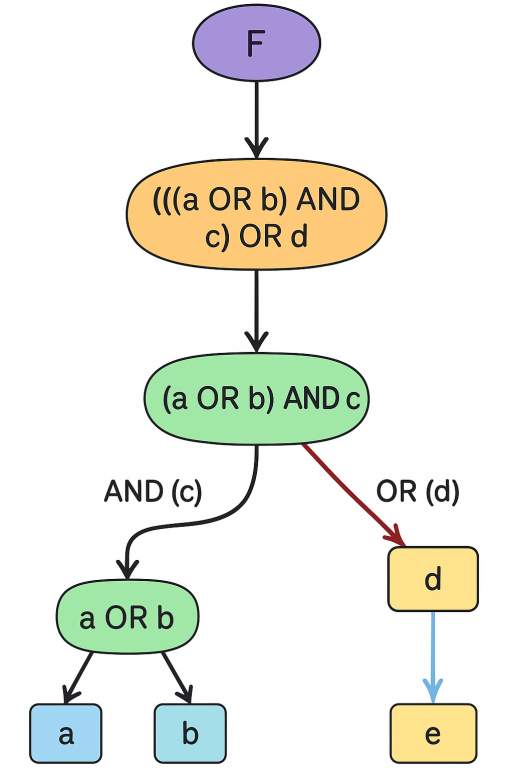
Variable: b
  0 → 1 changes (a OR b) → ... → F

Variable: c
  0 → 1 changes (a AND b) → ... → F

Variable: d
  0 → 1 changes OR d → ... → F

Variable: e
  0 → 1 changes final AND e → F

This tree shows **exactly why each variable can independently change F**, which is the essence of **MCDC**.



P6.5: States of Water (H2O)

Water can be at three states on the earth: solid, liquid, or gas. Let's denote them as S, L, and G, respectively. Transformation of states can happen under various conditions (inputs).
A software program is written to simulate water state changes through physical processes, with the following actions and effects:
(1) heating: S -> L; L -> G; S -> S; L -> L; G -> G.
(2) cooling: G -> L; L -> S; S -> S; L -> L; G -> G.
(3) precipitation: G -> L; G -> S.

(4) evaporation: L -> G.
(5) condensation: G -> L.
(6) deposition: G -> S.
(7) sublimation: S -> G.
Write down the finite state machine model for this system, plot the state transition diagram, and generate transaction coverage test cases from it.

1. Define the FSM Components

- **States (Q):** { S (solid), L (liquid), G (gas) }
- **Inputs (Σ):** { heating, cooling, precipitation, evaporation, condensation, deposition, sublimation }
- **Transition Function (δ):** maps (state, input) → next state
- **Start State:** Could be any of {S, L, G} (depends on simulation start)
- **Final States:** Not explicitly needed here, since it's a continuous physical process

2. Transition Table

Current State	Input	Next State	Notes
S	heating	L	Melting
S	cooling	S	No effect
S	precipitation	S	No effect
S	evaporation	S	No effect
S	condensation	S	No effect
S	deposition	S	No effect
S	sublimation	G	Sublimation
L	heating	G	Boiling
L	cooling	S	Freezing
L	precipitation	L	No effect
L	evaporation	G	Evaporation
L	condensation	L	No effect

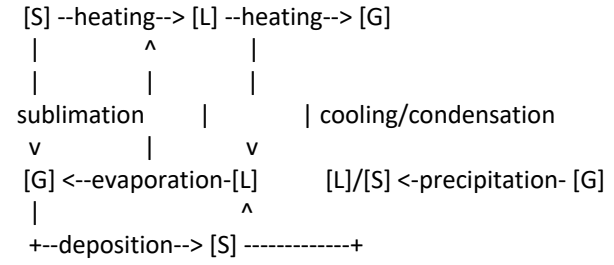
Current State	Input	Next State	Notes
L	deposition	L	No effect
L	sublimation	L	No effect
G	heating	G	No effect
G	cooling	L	Condensation/freezing
G	precipitation	L or S	Rain/snow (nondeterministic)
G	evaporation	G	No effect
G	condensation	L	Condensation
G	deposition	S	Frost (deposition)
G	sublimation	G	No effect

Note: “Precipitation” is nondeterministic: from Gas → Liquid **or** Solid. In FSM modeling, this means two possible transitions.

3. State Transition Diagram (Description)

- **Solid (S):**
  - heating → L
  - sublimation → G
- **Liquid (L):**
  - heating → G
  - cooling → S
  - evaporation → G
- **Gas (G):**
  - cooling → L
  - condensation → L
  - deposition → S
  - precipitation → L or S

Diagram (textual form):



4. Test Cases for Transaction (Transition) Coverage

Goal: Ensure each possible state transition is tested at least once.

From Solid (S):

1. S + heating → L
2. S + sublimation → G
3. S + cooling → S (self-loop)

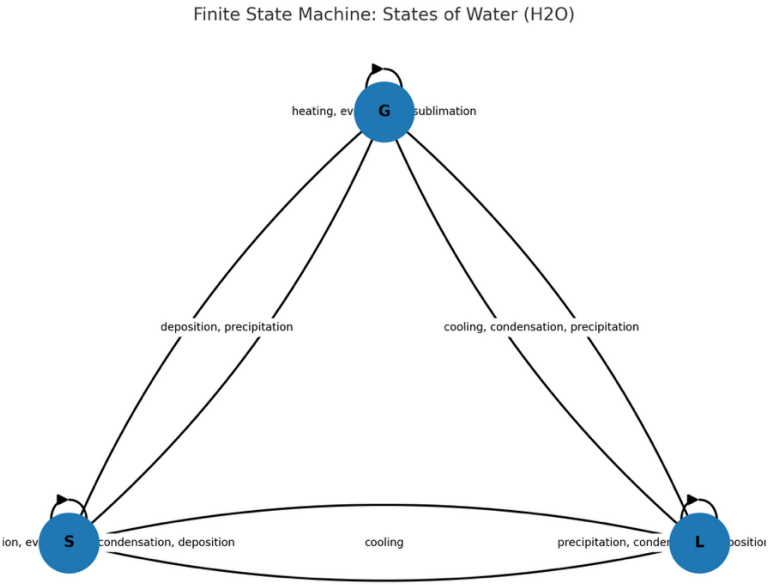
From Liquid (L):

4. L + heating → G
5. L + cooling → S
6. L + evaporation → G
7. L + precipitation/condensation/deposition/sublimation → L (self-loops)

From Gas (G):

8. G + cooling → L

9. G + condensation → L
10. G + deposition → S
11. G + precipitation → L
12. G + precipitation → S
13. G + heating → G (self-loop)



P6.6*. Quantum Physics: Schrödinger's Cat

Schrödinger's cat is a thought experiment. A cat is imagined as being enclosed in a box with a radioactive source and a poison that will be released when the source (unpredictably) emits radiation. If the poison is released the cat will be dead, but there is no way to tell if

the poison is released or not without opening the box and check it. More formally, the cat is being considered (according to quantum mechanics) to be simultaneously both dead and alive until the box is opened, and the cat is observed. The probability of the radioactive source emitting radiation is p.

Can FSM be used for this system? Is the system testable? What test cases you can come up with for this system if it can be tested? Note, in automata theory, this is an example of a nondeterministic finite automaton.

1. Can an FSM be used?

Yes.
This scenario can be modeled as a finite state machine, but **not** as a purely deterministic DFA; it is naturally a **nondeterministic finite automaton (NFA)** or better yet a **probabilistic finite automaton / probabilistic state machine** (PFA/Markov model). The nondeterminism / probability captures the radioactive emission event that may or may not occur (with probability ppp) while the cat is unobserved.

The important conceptual point: the quantum superposition ("dead *and* alive") is a physical interpretation — the FSM models the *observable classical outcomes* and the probabilistic branching that leads to those outcomes. You cannot directly observe the superposition without performing a measurement (opening the box), which collapses the system to one of the classical states.

2. FSM specification

States:

- UUU — *Unobserved* (cat inside box; emission may or may not have occurred)
- AAA — *Alive* (classical state: alive)
- DDD — *Dead* (classical state: dead)

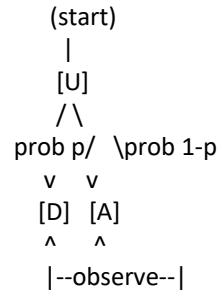
Inputs / events:

- wait or tick — time passes (during which emission might occur)
- emit — radioactive emission (internal event; occurs with probability ppp during the waiting window)
- observe or open_box — measurement/observation that reveals the cat's state

Transition function (informal):

- Start state: UUU.
- From UUU:
  - with probability ppp: U→emitDU \xrightarrow{\text{emit}} DUemitD
  - with probability 1−p1-p1−p: U \xrightarrow{\text{no_emit}} A
  - observe while in UUU causes collapse to whichever branch has occurred (equivalently, observe samples the emission nondeterministically/probabilistically with probability ppp for Dead).
- From AAA or DDD: observe yields that same classical outcome (already collapsed).

Compact ASCII diagram



Remarks:

- If you want to model repeated time ticks, you can view wait as generating an emit with probability ppp per waiting window.
- This is an NFA/PFA because from UUU there are two possible next states (A or D) and the choice is probabilistic.

3. Is the system testable?

Two distinct meanings of "testable":

1. **The physical thought experiment:** You cannot directly "test" or observe the superposition—any measurement collapses it. So you cannot *directly* show the cat was simultaneously alive and dead. Experimental verification of quantum superposition requires carefully controlled quantum systems and indirect interference/entanglement experiments — not a macroscopic cat experiment.
2. **A software model / simulator of the experiment:** Absolutely testable. You can:

- Unit-test deterministic behavior by *forcing* emission/no-emission (mock the RNG).
- Statistically test the probabilistic behavior (Monte-Carlo / hypothesis testing) to verify outcome frequencies match probability ppp.

So: the *model* and *simulator* are testable; the metaphysical notion of simultaneously alive & dead is not directly testable by a single observation.

4. Test cases

Below I list concrete test cases to **cover transitions** and to **verify probabilistic correctness**.

A. Transition coverage / deterministic unit tests (force RNG)

These tests assume you can control randomness (dependency-inject RNG or mock it).

1. **Forced emission**
   - Initial state: UUU.
   - Force emit = true.
   - Sequence: emit → observe.
   - Expected: Dead.
   - Purpose: verify U→emitDU \xrightarrow{\text{emit}} DUemitD and observation handling.
2. **Forced no-emission**
   - Initial state: UUU.
   - Force emit = false (or simulate no emission during waiting window).

- Sequence: wait (no emit) → observe.
- Expected: Alive.
- Purpose: verify U→no_emitAU \xrightarrow{\text{no_emit}} AUno_emitA.

3. **Immediate observation before emission window**
   - Start UUU. Call observe immediately (before any emission can occur).
   - Expected: Alive if the model defines the cat as initially alive and emission requires a time window.
   - Purpose: check event ordering and timing semantics.
4. **Observation is sticky**
   - Force no_emit and observe → Alive.
   - Then wait more time.
   - Expected: remains Alive (no retroactive change). Purpose: observation collapses state and prevents later reinterpretation within the same trial.
5. **Repeated observe idempotency**
   - After observe produces Alive or Dead, repeated observe operations should return the same outcome.

B. Probabilistic / statistical tests (Monte Carlo)

Test that the fraction of Dead outcomes matches ppp within statistical tolerance.

Procedure (standard):

- Choose probability ppp.
- Choose number of trials NNN.

- Run NNN independent simulations of the experiment (each trial: run waiting window, then observe).
- Let p^\hat{p}p^ be observed fraction of Dead.
- Use binomial statistics to check whether p^\hat{p}p^ is consistent with ppp.

Worked numeric example (explicit arithmetic step-by-step)

Let p=0.20p = 0.20p=0.20 and N=1000N = 1000N=1000.

1. Compute variance of p^\hat{p}p^: Var(p^)=p(1−p)N\mathrm{Var}(\hat{p}) = \dfrac{p(1-p)}{N}Var(p^)=Np(1−p).
    - p(1−p)=0.20×0.80=0.160p(1-p) = 0.20 \times 0.80 = 0.160p(1−p)=0.20×0.80=0.160.
    - p(1−p)N=0.1601000=0.000160\dfrac{p(1-p)}{N} = \dfrac{0.160}{1000} = 0.000160Np(1−p) =10000.160=0.000160.
2. Standard error σ=p(1−p)N=0.000160\sigma = \sqrt{\dfrac{p(1-p)}{N}} = \sqrt{0.000160}σ=Np(1−p) =0.000160.
    - σ≈0.0126491106406735 2\sigma \approx 0.0126491106406735 2σ≈0.0126491106406735 2. (computed by square-root)
3. Use a 3σ acceptance band (≈99.7% normal-approx coverage):
    - 3σ≈3×0.0126491106406735 2≈0.037947331922020563\sigma \approx 3 \times 0.0126491106406735 2 \approx 0.037947331922020563σ≈3×0.0126491106406735 2≈0.03794733192202056.
4. Acceptance interval for observed proportion:
    - Lower: p−3σ=0.20−0.03794733192202056≈0.162052668

07797944p - 3\sigma = 0.20 - 0.03794733192202056 \approx 0.16205266807797944p−3σ=0.20−0.03794733192202056≈0.16205266807797944.

- Upper: p+3σ=0.20+0.03794733192202056≈0.23794733192202056p + 3\sigma = 0.20 + 0.03794733192202056 \approx 0.23794733192202056p+3σ=0.20+0.03794733192202056≈0.23794733192202056.

5. Translate to counts (out of 1000):
   - Lower count ≈ 0.16205×1000≈162.050.16205 \times 1000 \approx 162.050.16205×1000≈162.05 → round to 162.
   - Upper count ≈ 0.23795×1000≈237.950.23795 \times 1000 \approx 237.950.23795×1000≈237.95 → round to 238.

Acceptance rule: For this test, pass if the number of Dead outcomes is between **162 and 238** (inclusive). If outside this range, suspect bug or RNG problem. For small NNN use exact binomial tests; for larger NNN this normal approximation is OK.

Other recommended statistical checks:

- Repeat for multiple ppp values (0.0, 0.01, 0.5, 0.9, 1.0).
- Use exact binomial test or chi-square goodness-of-fit when appropriate.
- Use random seeds to reproduce failing runs.

C. Boundary & edge cases

- p=0.0p = 0.0p=0.0: expect **0 dead** in any run. (Strict test: all trials alive.)

- p=1.0p = 1.0p=1.0: expect **all dead**. (Strict test: all trials dead.)
- Very small ppp (e.g. 10−610^{-6}10−6) to test numerical stability (requires very many trials to test statistically).
- Test what happens if observe is called repeatedly, or if observe can be called concurrently in multi-threaded simulation (test thread-safety invariants).

D. Property-based / invariants

- After observe, the internal observed flag is true.
- After observe, result is stable (re-observe returns same value).
- Number of emitted events per trial is 0 or 1 depending on model semantics.

5. Small reference simulator (Python)

Below is a tiny deterministic/probabilistic simulator you can paste and run locally. It supports forcing emission for unit tests and running Monte Carlo trials.

```
import random

class SchrodingerSimulator:
    def __init__(self, p, rng=None):
        self.p = p
        self.rng = rng or random.Random()

    def run_trial(self, force_emit=None):
        """
        Run a single trial.
        - force_emit: None (random), True (force emission), False (force no emission)
        Returns: 'Dead' or 'Alive'
        """
```

```
        if force_emit is True:
            emitted = True
        elif force_emit is False:
            emitted = False
        else:
            emitted = (self.rng.random() < self.p)

        # observation collapses:
        return "Dead" if emitted else "Alive"

    def monte_carlo(self, N):
        dead_count = 0
        for _ in range(N):
            if self.run_trial() == "Dead":
                dead_count += 1
        return dead_count, dead_count / N

# Example usage:
if __name__ == "__main__":
    sim = SchrodingerSimulator(p=0.2)
    # Deterministic tests:
    assert sim.run_trial(force_emit=True) == "Dead"
    assert sim.run_trial(force_emit=False) == "Alive"

    # Monte Carlo:
    dead_count, prop = sim.monte_carlo(1000)
    print("Dead count:", dead_count, "Fraction:", prop)
```

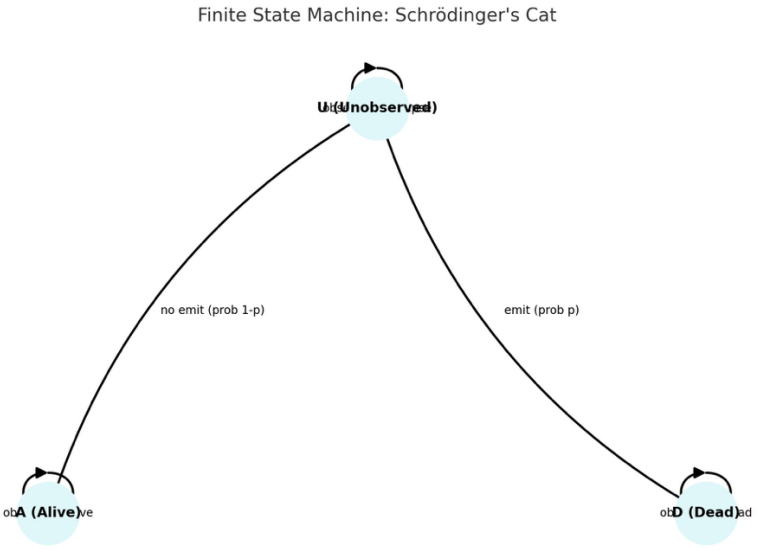
Use the deterministic force_emit parameter to implement unit tests that validate transition logic. Use monte_carlo(N) and the statistical acceptance rules above to validate probabilistic behavior.

6. Final remarks / interpretation

- The FSM/PFA is a perfectly valid formal model of the experiment's *observable outcomes* and their probabilities. In automata theory the branch from UUU

to AAA or DDD is nondeterministic (or probabilistic if you attach probability ppp).

- Testing focuses on (a) correctness of the model/simulator and (b) statistical agreement with the specified ppp.
- Ethically: Schrödinger's cat is only a thought experiment — use simulation and don't attempt harmful physical experiments.



Python implementation:

See ***test_schrodinger_simulator.py*** under scripts at the this repo.

Chapter 7 GENERAL PRINCIPLES OF TEST CASE DESIGN

P7.1: Answer the following questions based on the discussions in this chapter.

(1) Explain the atomic principle.
(2) Explain the independency principle.
(3) What are the five categories of independencies? Explain each of them.
(4) Explain the modularization principle. Give an example of testing utilizing the modularization principle.
(5) What's the HFSM model? And how it's applied to software testing?
(6) Give an example of manual testing using the modularization principle.
(7) Explain the test coverage principle.

Solutions:

(1) The Atomic Principle

The **atomic principle** in software testing states that a **test case should be minimal, indivisible, and focus on a single objective**.

- Like an atom, it cannot be meaningfully broken down further.
- Each atomic test case should check **one specific functionality, path, or requirement**, so that its outcome (pass/fail) is unambiguous.

Example:
Instead of writing one large test that checks login, dashboard rendering, and logout together, an atomic test would separately check:

- entering valid credentials → login succeeds,
- entering invalid credentials → login fails,
- successful logout → user session ends.

(2) The Independency Principle

The **independency principle** requires that **each test case can be executed independently of others**, in any order, without being affected by prior tests.

- Independence ensures **repeatability, reliability, and isolation of failures**.
- It prevents "chain reaction" errors where one failing test causes others to fail, even if their underlying functionality is correct.

(3) The Five Categories of Independencies

The five categories are:

1. **Input Independency**
    - Test inputs should not depend on results of other tests.
    - Example: Each login test provides its own username/password instead of reusing data left by earlier tests.

2. **Output Independency**
    - Test outputs/results should be verifiable without relying on other test outcomes.
    - Example: A file-read test checks expected content in isolation, not assuming another test already wrote it.
3. **Environment Independency**
    - A test should not assume a specific environmental condition set up by a previous test.
    - Example: A web test shouldn't assume the user is already logged in unless it performs the login itself.
4. **Execution Independency**
    - Test cases should be runnable in any order (or in parallel).
    - Example: Running Test B before Test A should not change results.
5. **Result Independency**
    - A test's pass/fail decision must be made solely on its own expected vs. actual result.
    - Example: Unit test for "addition function" passes/fails based on its own assertion, not influenced by prior unrelated tests.

---

(4) The Modularization Principle

The **modularization principle** means that testing should follow the system's modular design: break testing tasks into smaller, manageable **modules or components**, each testable independently before integration.

- This makes testing **scalable, maintainable, and easier to debug**.

Example:
In an e-commerce system:

- Test the **shopping cart module** separately (add/remove item, update quantity).
- Test the **payment module** separately (valid/invalid credit card).
- Then perform **integration testing** across modules (cart → checkout → payment).

(5) The HFSM Model

The **HFSM (Hierarchical Finite State Machine)** model extends the traditional FSM by introducing **hierarchical states** and **nested transitions**.

- Complex systems can be decomposed into **higher-level states**, each containing **sub-states**.
- This avoids "state explosion" and provides **structured test modeling**.

Application in software testing:

- HFSMs are used in **model-based testing**: representing software behavior at multiple abstraction levels.
- Test cases are derived systematically by traversing states and transitions.
- Example: A mobile banking app → Top-level states: Login, Transaction, Logout. Inside Transaction: sub-states Transfer, Bill

Payment, Deposit. Tests can focus on each sub-state while still covering transitions between top-level states.

(6) Example of Manual Testing using Modularization Principle

Scenario: Testing a **Library Management System** manually.

- **Module 1: User Registration**
  - Test: Register a new user with valid details → success.
  - Test: Register with invalid email → error message.
- **Module 2: Book Search**
  - Test: Search by title → correct book list returned.
  - Test: Search with nonexistent keyword → "no results" message.
- **Module 3: Borrow/Return**
  - Test: Borrow an available book → book marked as checked out.
  - Test: Return book → marked available again.

Here, **each module is tested independently first**, then integration testing ensures smooth interaction (e.g., registered user borrowing books).

(7) The Test Coverage Principle

The **test coverage principle** requires that testing should **quantitatively measure how much of the software has been exercised**, ensuring completeness and guiding further testing.

- Coverage metrics can include:
    - **Code coverage** (statements, branches, paths)
    - **Requirement coverage** (every requirement has at least one test)
    - **State/transition coverage** (for FSM/HFSM models)
- High coverage reduces risk of untested parts of the system hiding defects.

Example:
In unit testing: achieving **branch coverage** means that for every `if-else` condition, both the `true` and `false` branches are tested.

Chapter 8 – Regression Testing

P8.1 Short answers

1. **Examples:**
    - After refactoring a core library, re-run unit and API test suites that cover impacted modules.
    - After fixing a bug in order totals, re-run UI checkout, pricing, and tax calculation tests.
2. **Why needed:** Code/dep changes can **reintroduce defects** or break existing behavior; regression keeps quality stable across releases/CI cycles.

3. **Major change types requiring regression:** Feature changes, bug fixes, configuration/infra upgrades, dependency/library updates, schema changes, and performance-sensitive rewrites.
4. **Selection techniques:**
    - **Retest-all** (max safety, high cost)
    - **Selective retest based on impact analysis** (change-based, covers touched areas and dependencies)
    - **Test case prioritization** (run high-value tests first—e.g., risk, business criticality, historical failure data)
    - **Hybrid approaches** combining the above
5. **Prioritization factors:** Business importance, change risk/size, historical defect density, execution time/cost, and coverage gaps—often combined via a **weighted score** to scale to P0/P1/P2/P3 tiers (see example tables).
6. **Keeping suites up to date:** Modify tests after functionality changes, remove obsolete/redundant tests, repair flaky/expensive tests, and integrate into **CI** for continual health checks.

Chapter 9 UNIT TESTING (TYPE 0 TESTS)

P9.1: Seasons of The Year

The following code segment is a Java implementation of a getSeason() method, generate test cases using any technique, and write automated unit tests (in any testing framework or language you prefer).

```
public static String getSeason (int month) {
switch(month) {
case 12:
case 1:
case 2:
return "Winter";
case 3:
case 4:
case 5:
return "Spring";
case 6:
case 7:
case 8:
return "Summer";
case 9:
case 10:
case 11:
return "Autumn";
default:
throw new IllegalArgumentException("A month is a number
from
```

```
        1 to 12.");
}
}
```

Step 1: Equivalence Classes

The function accepts **months (1–12)** → returns a season, otherwise throws an exception.
So equivalence classes are:

- **Valid months:**
  - Winter: {12, 1, 2}
  - Spring: {3, 4, 5}
  - Summer: {6, 7, 8}
  - Autumn: {9, 10, 11}
- **Invalid months:**
  - Less than 1 (e.g., 0, -1)
  - Greater than 12 (e.g., 13, 100)

Step 2: Boundary Values

- Around 1: (0, 1, 2)
- Around 12: (11, 12, 13)
- Mid-values in each season: (4, 7, 10)

Step 3: Test Cases

Test ID	Input	Expected Output
T1	1	"Winter"
T2	2	"Winter"

Test ID	Input	Expected Output
T3	12	"Winter"
T4	3	"Spring"
T5	5	"Spring"
T6	6	"Summer"
T7	8	"Summer"
T8	9	"Autumn"
T9	11	"Autumn"
T10	0	Exception
T11	-5	Exception
T12	13	Exception
T13	100	Exception

This covers **all partitions + boundaries**.

Step 4: Automated Unit Tests (pytest)

Here's a **Python version** of the unit tests that replicates the Java logic:

```
import pytest

def get_season(month: int) -> str:
    if month in (12, 1, 2):
        return "Winter"
    elif month in (3, 4, 5):
        return "Spring"
    elif month in (6, 7, 8):
        return "Summer"
    elif month in (9, 10, 11):
        return "Autumn"
    else:
```

```
        raise ValueError("A month is a number from 1 to 12.")

# ---------------- Unit Tests ---------------- #

@pytest.mark.parametrize("month, expected", [
    (1, "Winter"), (2, "Winter"), (12, "Winter"),
    (3, "Spring"), (5, "Spring"),
    (6, "Summer"), (8, "Summer"),
    (9, "Autumn"), (11, "Autumn"),
])
def test_valid_months(month, expected):
    assert get_season(month) == expected

@pytest.mark.parametrize("month", [0, -5, 13, 100])
def test_invalid_months(month):
    with pytest.raises(ValueError):
        get_season(month)
```

JUnit Test Class

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class SeasonTest {

    // Reference implementation (from the problem statement)
    public static String getSeason(int month) {
        switch(month) {
            case 12: case 1: case 2:
                return "Winter";
            case 3: case 4: case 5:
                return "Spring";
            case 6: case 7: case 8:
                return "Summer";
            case 9: case 10: case 11:
                return "Autumn";
            default:
                throw new IllegalArgumentException("A month is a number
from 1 to 12.");
```

```
    }
  }

  // ---- Valid Month Tests ----
  @Test
  public void testWinterMonths() {
    assertEquals("Winter", getSeason(1));
    assertEquals("Winter", getSeason(2));
    assertEquals("Winter", getSeason(12));
  }

  @Test
  public void testSpringMonths() {
    assertEquals("Spring", getSeason(3));
    assertEquals("Spring", getSeason(5));
  }

  @Test
  public void testSummerMonths() {
    assertEquals("Summer", getSeason(6));
    assertEquals("Summer", getSeason(8));
  }

  @Test
  public void testAutumnMonths() {
    assertEquals("Autumn", getSeason(9));
    assertEquals("Autumn", getSeason(11));
  }

  // ---- Invalid Month Tests ----
  @Test
  public void testInvalidMonthZero() {
    assertThrows(IllegalArgumentException.class, () ->
getSeason(0));
  }

  @Test
  public void testInvalidNegativeMonth() {
    assertThrows(IllegalArgumentException.class, () -> getSeason(-
5));
  }
```

```
   @Test
   public void testInvalidMonthGreaterThan12() {
      assertThrows(IllegalArgumentException.class, () ->
getSeason(13));
      assertThrows(IllegalArgumentException.class, () ->
getSeason(100));
   }
}
```

P9.2: Right Triangle

The following code segment is a Java implementation of a method to check if a triangle with three integer sides is a right triangle. Generate test cases using any technique discussed in part 2 of the book and write automated unit tests.

```
public static boolean isRightTriangle (int a, int b, int c) {
if ( (a<=0) || (b<=0) || (c<=0) ) {
System.out.println ("All sides are positive numbers.");
return false;
}
return (Math.pow(a, 2) + Math.pow(b, 2) == Math.pow(c, 2))
||
(Math.pow(a, 2) + Math.pow(c, 2) == Math.pow(b, 2)) ||
(Math.pow(c, 2) + Math.pow(b, 2) == Math.pow(a, 2));
}
```

Testing Techniques Applied:

1. Equivalence Partitioning (EP)

- Valid right triangles with different Pythagorean triples
- Invalid triangles that aren't right triangles
- Invalid inputs (negative and zero values)

2. Boundary Value Analysis (BVA)

- Tests around the zero boundary (0, -1, 1)
- Minimum valid Pythagorean triple (3,4,5)
- Large values to test for potential overflow

3. Decision Table Testing (DT)

- Systematic testing of all condition combinations:
    - All sides positive: True/False
    - Pythagorean theorem satisfied: True/False

4. Pairwise Testing (PW)

- Tests all parameter orderings for the same triangle values
- Ensures the method works regardless of parameter order

5. Special Cases and Edge Cases

- Equal sides (equilateral triangles)
- Two equal sides (isosceles triangles)
- Degenerate cases that violate triangle inequality

6. Precision Testing

- Tests integer precision with known Pythagorean triples
- Near-miss cases that are close but not exact right triangles

7. Regression Testing

- Comprehensive list of known Pythagorean triples
- Scaled versions of primitive triples

Key Test Categories:

- **Positive Tests**: Valid right triangles with various Pythagorean triples
- **Negative Tests**: Invalid inputs and non-right triangles
- **Boundary Tests**: Values at the edges of valid ranges
- **Error Condition Tests**: Multiple invalid conditions simultaneously
- **Stress Tests**: Large values to test robustness

Issues Identified in the Original Code:

1. **Floating-point precision**: Using `Math.pow()` and `==` for floating-point comparison could lead to precision issues
2. **Error message**: The error message says "All sides are positive numbers" but should say "All sides must be positive numbers"
3. **No triangle inequality check**: The method doesn't verify if the three sides can actually form a triangle

The test suite includes 67 individual test cases covering all major scenarios and edge cases, ensuring comprehensive validation of the isRightTriangle method functionality.

The java code is at **test_triangle.java**.

P9.3: Prime Numbers

The following code segment is a Java implementation of a method to check if a number is a prime number, generate test cases with equivalence partitioning technique, and write automated unit tests. Sub-partitioning, 2nd level partitions within a partition, should be considered in certain partitions. Any other concerns?

```
public static boolean isPrimeNumber (int n) {
if (n <= 1)
return false;
for (int i = 2; i < Math.sqrt(n); i++) {
if (n % i == 0)
return false;
}
return true;
}
```

Solution:

This implementation has a critical bug: the loop condition should be `i <= Math.sqrt(n)` instead of `i < Math.sqrt(n)`. This means the method will incorrectly identify perfect squares of prime numbers (like 4, 9, 25, 49, 121) as prime numbers because it doesn't check the square root as a potential divisor.

Equivalence Partitioning with Sub-Partitioning

1. Invalid Input Partitions

- **Partition 1:** Negative numbers (n < 0)

- **Partition 2:** Zero (n = 0)

- **Partition 3:** One (n = 1)

2. Valid Input Partitions (Primes)

- **Partition 4:** The only even prime (n = 2)

- **Partition 5:** Small odd primes (3 ≤ n < 100)

- **Partition 6:** Large odd primes (n ≥ 100)

3. Valid Input Partitions (Non-Primes/Composites)

- **Partition 7:** Small even composites (4 ≤ n < 100, n even)

- **Partition 8:** Small odd composites (9 ≤ n < 100, n odd)

- **Partition 9:** Large even composites (n ≥ 100, n even)

- **Partition 10:** Large odd composites (n ≥ 100, n odd)

- **Partition 11:** Perfect squares (n = k^2 where k > 1) - *Critical sub-partition due to bug*

Test Cases for Each Partition

| Partition | Test Case | Expected Result | Notes |

|---|---|---|---|

| 1 | -5 | false | Negative number |

| 1 | -100 | false | Negative number |

| 2 | 0 | false | Zero |

| 3 | 1 | false | One |

| 4 | 2 | true | Only even prime |

| 5 | 3 | true | Small odd prime |

| 5 | 7 | true | Small odd prime |

| 5 | 97 | true | Small odd prime |

| 6 | 101 | true | Large prime |

| 6 | 997 | true | Large prime |

| 6 | 2147483647 | true | Integer.MAX_VALUE (prime) |

| 7 | 4 | false | *Will fail due to bug* (smallest composite, perfect square) |

| 7 | 6 | false | Small even composite |

| 7 | 96 | false | Small even composite |

| 8 | 9 | false | *Will fail due to bug* (small odd composite, perfect square) |

| 8 | 15 | false | Small odd composite |

| 8 | 95 | false | Small odd composite |

| 9 | 100 | false | Large even composite |

| 9 | 102 | false | Large even composite |

| 10 | 105 | false | Large odd composite |

| 10 | 1001 | false | Large odd composite |

| 11 | 25 | false | *Will fail due to bug* (perfect square) |

| 11 | 49 | false | *Will fail due to bug* (perfect square) |

| 11 | 121 | false | *Will fail due to bug* (perfect square) |

JUnit Test Implementation

See **test_prime_number.java**

Critical Observations and Recommendations

1. **The Bug**: The implementation incorrectly uses `i < Math.sqrt(n)` instead of `i <= Math.sqrt(n)`, causing perfect squares of primes to be misidentified as prime numbers.

2. **Sub-Partitioning Importance**: This problem perfectly illustrates why sub-partitioning is crucial. The perfect squares sub-partition (Partition 11) revealed a critical bug that standard partitioning might have missed.

3. **Optimization Opportunity**: The current implementation checks even numbers unnecessarily after handling n=2. An optimized version would:

   - Immediately return false for even numbers > 2

   - Only check odd divisors starting from 3

   - Use the correct loop condition `i <= Math.sqrt(n)`

4. **Boundary Analysis**: For large numbers near Integer.MAX_VALUE, the method could face performance issues. Consider using more efficient primality tests for very large numbers.

5. **Test Case Prioritization**: Given the bug affects perfect squares, tests for perfect squares should be prioritized in the regression test suite.

This comprehensive test suite not only validates the current implementation but also identifies and documents the critical bug, providing clear evidence of where the implementation fails

and why. The tests are designed to be atomic, independent, and to follow best practices for unit testing.

P9.4: Bank Account Balance Updates

The following code segment shows a bank account update method that takes a customer ID and a deposit amount and updates the total balance in the database. The DatabaseInterface interface is defined with two methods, one to. get the current balance, and the other to update the balance.

```
public class BalanceUpdateService {
private final DatabaseInterface dbaccess;
public BalanceUpdateService (DatabaseInterface dbaccess) {
this.dbaccess = dbaccess;
}
public void UpdateAccountBalance (int customer_id, float
deposit) {
if ((deposit <= 0.0) || customer_id <= 0)) {
// error message here
}
else {
float balance = dbaccess.getBalance (customer_id);
dbaccess.updateBalance (customer_id, balance + deposit);
}
}
}
public interface DatabaseInterface {
public float getBalance (int customer_id);
public void updateBalance (int customer_id, float deposit);
}
```

Write unit tests using mocking and stubbing techniques for test cases with

valid input vectors (in the valid partition).

Solution:

Key Testing Techniques Used:

1. Mockito Mocking Framework

- @Mock annotation to create mock DatabaseInterface
- MockitoAnnotations.openMocks() for initialization
- Proper mock lifecycle management with @BeforeEach

2. Stubbing Techniques

- **Simple Stubbing**: when().thenReturn() for basic method return values
- **Conditional Stubbing**: Different return values for different customer IDs
- **Reset and Re-stub**: Testing multiple scenarios with reset()

3. Verification Techniques

- **Interaction Verification**: verify() to ensure methods are called
- **Parameter Verification**: ArgumentCaptor to verify exact parameters passed
- **Sequence Verification**: InOrder to verify method call sequence
- **Interaction Count**: times(), never(), only() verification

- **No More Interactions**: verifyNoMoreInteractions() for complete verification

Test Categories (Valid Input Focus):

Basic Functionality Tests

- Small deposits to zero balance accounts
- Large deposits to existing balance accounts
- Minimum boundary values (just above zero)
- High customer IDs with decimal amounts

Different Balance Scenarios

- Accounts with high existing balances
- Multiple deposit simulations
- Overdraft scenarios (negative balances)

Precision and Edge Cases

- Deposits with many decimal places
- Very small deposit amounts
- Float arithmetic handling with tolerance

Interaction Verification Tests

- Correct sequence of database calls
- Exact number of interactions
- Parameter passing verification
- No side effects verification

Stubbing Variations

- Different stub return values
- Conditional stubbing for multiple customers

- Reset and re-stub scenarios

Key Features of the Test Suite:

1. **Comprehensive Coverage**: 15 test methods covering various valid input scenarios
2. **Proper Mock Usage**: Demonstrates correct mocking patterns and best practices
3. **Argument Verification**: Uses ArgumentCaptor to verify exact values passed to mocked methods
4. **Float Precision Handling**: Includes delta comparisons for floating-point arithmetic
5. **Interaction Safety**: Verifies no unexpected method calls occur
6. **Real-world Scenarios**: Tests practical banking scenarios like overdrafts and multiple deposits

Issues Identified in Original Code:

1. **Syntax Error**: Missing opening parenthesis in the if condition
2. **Method Naming**: Should follow Java conventions (updateAccountBalance, not UpdateAccountBalance)
3. **Error Handling**: No exception throwing, only console output
4. **Float Precision**: Using float for money calculations may cause precision issues

The test suite ensures that the BalanceUpdateService correctly interacts with the database interface for all valid input scenarios while maintaining proper separation of concerns through mocking.

See bank_account_test.java under/scripts

Chapter 11 – Performance Testing

P11.1 Basics

- **Percentile vs average:** Percentiles (e.g., p95/p99) capture **tail latency** and user experience better than averages when distributions are skewed or multi-modal; averages can hide outliers.
- **Normal distributions:** Many natural/aggregate phenomena trend normal (e.g., measurement noise, aggregated small independent latencies), but **request latencies** often **aren't normal**—they're skewed, strengthening the percentile case.
- **Capacity dimensions:** Concurrency (users/threads), throughput (req/s), response time, resource utilization (CPU, memory, IO), and saturation/queueing behavior.
- **Major factors:** Code efficiency, DB queries/indexing, network, contention/locks, GC/tuning, caching, configuration, and hardware.
- **Main goals:** Verify **responsiveness**, **capacity**, **scalability**, and **stability** under expected and peak loads; find bottlenecks early.

P11.2 Types of Performance Testing

- **Benchmark:** Baseline performance comparisons of versions/builds.
- **Load:** Behavior under expected/gradual increasing load.
- **Stress:** Beyond capacity, to failure modes and recovery.
- (All are Type 2.* per the book's classification.)

P11.3 Processes

- **Steps:** Define goals/SLOs; model workload; design scenarios & datasets; configure environment; run tests (warm-up included); collect/validate metrics; analyze & tune; re-test.
- **Metrics:** Response time distribution (p50/p90/p95/p99/max), throughput, error rate, resource utilization (CPU/mem/disk/net), GC stats, queue lengths.

P11.4 Test Design & Validity

- **Design factors:** Realistic traffic mix, think time, data variability, cache warm/cold, ramp patterns, concurrency, test duration, and environment parity.
- **Validity:** Control for noise; warm-ups; repeatability; isolate external dependencies; verify no error masking; confirm steady-state; correlate app and infra metrics.