

Software Engineering and Testing (315332)

Model Question Paper — Set B · Model Answers

Q.1

a) Types of software.

2 M

1. **System software:** programs written to service other programs, e.g. operating system, compiler, drivers.
2. **Application software:** stand-alone programs that solve a specific business need, e.g. payroll, billing, railway reservation.
3. **Engineering / scientific software:** number-crunching and design programs, e.g. CAD, simulation, MATLAB.
4. **Embedded software:** resides in read-only memory of a product and controls it, e.g. washing-machine controller, car ABS.

(Any four, $\frac{1}{2}$ M each)

b) SRS and its needs.

2 M

SRS (Software Requirement Specification) is a document that completely describes the functional and non-functional requirements of the proposed software — *what* the system will do, not *how*.

1. **Agreement between customer and developer:** it records exactly what the software must do, so both sides have the same understanding.
2. **Reduces development effort and rework:** careful review of the SRS finds omissions and misunderstandings early, before design and coding.
3. **Basis for estimating cost and schedule:** size, effort and time are estimated from the documented requirements.

(Definition 1 M, two needs 1 M)

c) Risk and its classification.

2 M

Risk is a potential problem — an uncertain future event that may or may not happen but, if it happens, causes loss. Two characteristics: *uncertainty* and *loss*.

- **Project risks** (budget, schedule, staff, resources)
- **Technical risks** (design, implementation, interface, verification)
- **Business risks** (market, strategic, sales, management, budget)
- Also: **known, predictable** and **unpredictable** risks.

(Definition 1 M, classification 1 M)

d) Quality assurance and quality control.

2 M

Quality Assurance (QA): a planned and systematic set of activities (process definition, standards, audits, reviews) that gives confidence that the **process** used to build the software will produce a quality product. It is *preventive*.

Quality Control (QC): a set of activities (inspections, reviews, testing) carried out on the **product** to find defects and check that it meets its requirements before release. It is *detective / corrective*.

(Each definition 1 M)

e) Defect with example.

2 M

Defect: A defect is a **variance (deviation) from the desired product attribute** — i.e. a flaw in a work product (requirement, design or code) because of which the software does not meet its specification or the customer's expectation. A defect found during testing is commonly called a *bug*.

Example: the SRS says a customer gets 10 % discount on bills above ₹ 1000, but the program gives the discount only above ₹ 2000 — a *wrong* implementation of the specification.

(Definition 1 M, example 1 M)

f) System testing.

2 M

System testing is the testing of the **complete, integrated system** to verify that it meets its specified requirements (SRS). It is black-box testing done by an independent test team in an environment close to production and covers functional and non-functional requirements — recovery testing, security testing, stress testing, performance (load) testing, deployment (configuration / compatibility) testing, usability testing.

(Definition 2 M)

g) Objectives of project planning.

2 M

The **objective of project planning** is to provide a framework that enables the manager to make **reasonable estimates of resources, cost and schedule**. Estimates are made within a limited time frame at the start of the project and are updated regularly as the project progresses; they also define best-case and worst-case scenarios so that project outcomes can be bounded.

(Objective 2 M)

Q.2

a) Layers of software engineering.

4 M

Software engineering is a **layered technology**: any engineering approach (including software engineering) must rest on an organisational commitment to quality.

Tools
Methods
Process
A quality focus

Software engineering — a layered technology

1. **A quality focus:** the bedrock. Any engineering approach must rest on an organisational commitment to quality; total quality management, Six Sigma and similar philosophies foster continuous process improvement.
2. **Process:** the foundation layer and the glue that holds the technology layers together. It defines the framework of key process areas, the context in which methods are applied, work products are produced, milestones are established, quality is ensured and change is managed.
3. **Methods:** provide the technical “how-to’s” for building software — communication, requirements analysis, design modelling, program construction, testing and support.
4. **Tools:** provide automated or semi-automated support for the process and methods (CASE tools); when tools are integrated, information created by one tool is used by another.

Why “bedrock”: every layer above rests on the organisation’s commitment to quality; without a quality culture, process, methods and tools cannot produce good software.

(Diagram 1 M, layers 2 M, bedrock 1 M)

b) Waterfall model.

4 M

The **waterfall model** (classic life cycle) suggests a systematic, **sequential** approach: each phase is completed before the next one starts.

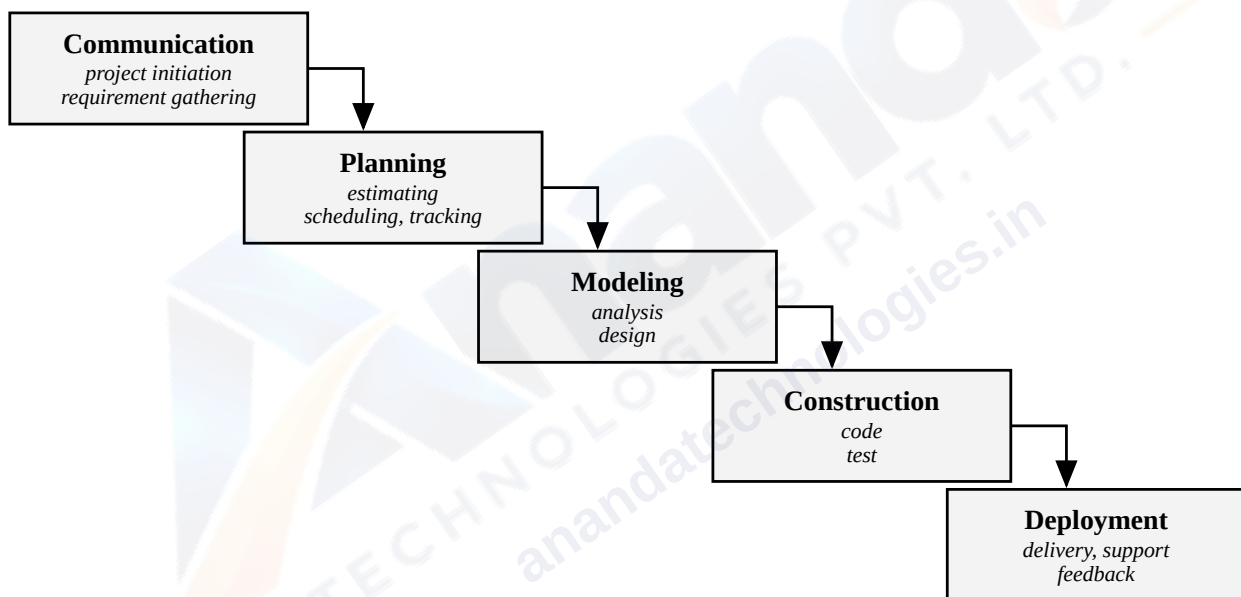


Fig.: Waterfall model (classic life cycle)

1. **Communication:** communicate and collaborate with the customer and stakeholders to understand objectives and gather requirements.
2. **Planning:** create a “map” (software project plan) — technical tasks, risks, resources, work products and schedule.
3. **Modeling:** create models (analysis and design) to understand requirements and the design that will achieve them.
4. **Construction:** code generation (manual or automated) and testing to uncover errors in the code.
5. **Deployment:** deliver the software (complete or increment) to the customer, who evaluates it and gives feedback.

Use: when requirements are well understood and stable. **Drawbacks:** real projects rarely follow a sequence; customer sees a working version only late; changes are costly.

(Diagram 2 M, explanation 2 M)

c) Adaptive Software Development.

4 M

Adaptive Software Development (ASD), proposed by Jim Highsmith, is an agile technique for building complex software and systems. Its focus is on human collaboration and team self-organisation. The ASD life cycle has three phases:

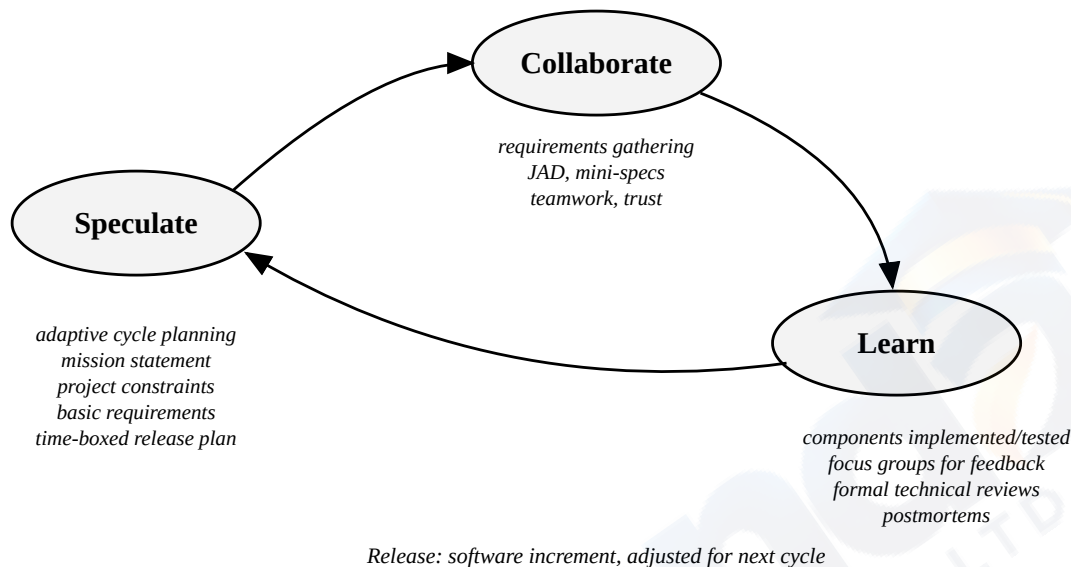


Fig.: Adaptive Software Development (ASD) life cycle

i) Speculation: the project is initiated and *adaptive cycle planning* is conducted. It uses project-initiation information — the customer’s *mission statement*, *project constraints* (delivery dates, user descriptions) and *basic requirements* — to define the set of release cycles (software increments). The plan is reviewed after each cycle and adjusted to match reality.

ii) Collaboration: motivated people work together in a way that multiplies their talent. It requires *trust* — team members criticise without animosity, assist without resentment, work as hard as or harder than they do, have the skills to contribute, and communicate problems effectively. Requirements are gathered with JAD sessions and mini-specs.

iii) Learning: as the team builds components, the emphasis is on learning — through *focus groups* (customer feedback on increments), *formal technical reviews* and *postmortems* (the team reviews its own performance and process).

(Diagram 1 M, three phases 3 M)

d) Level 0 and level 1 DFD — Hostel Management System.

4 M

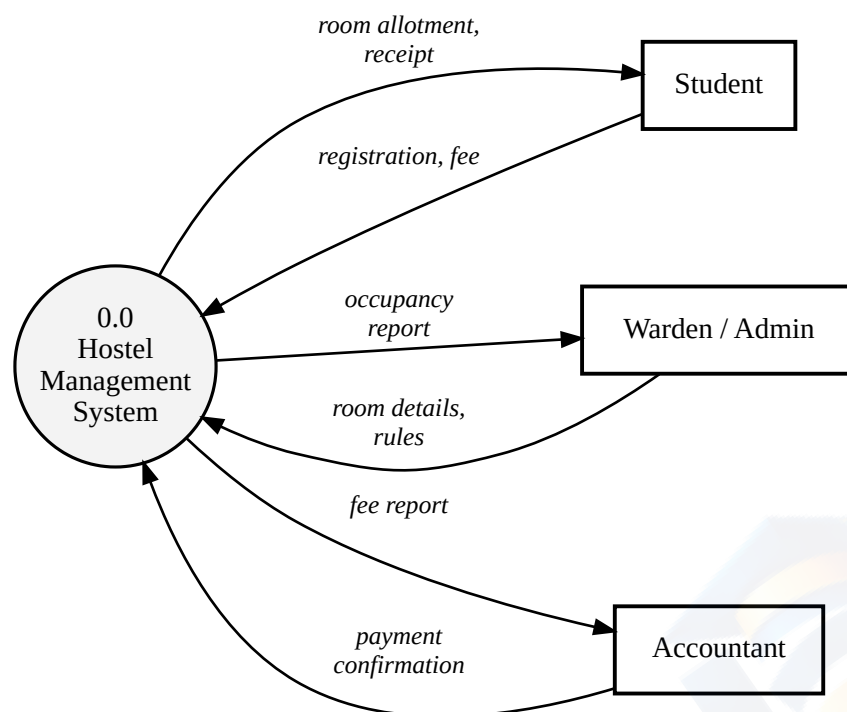


Fig.: Level 0 DFD (context diagram) — Hostel Management System

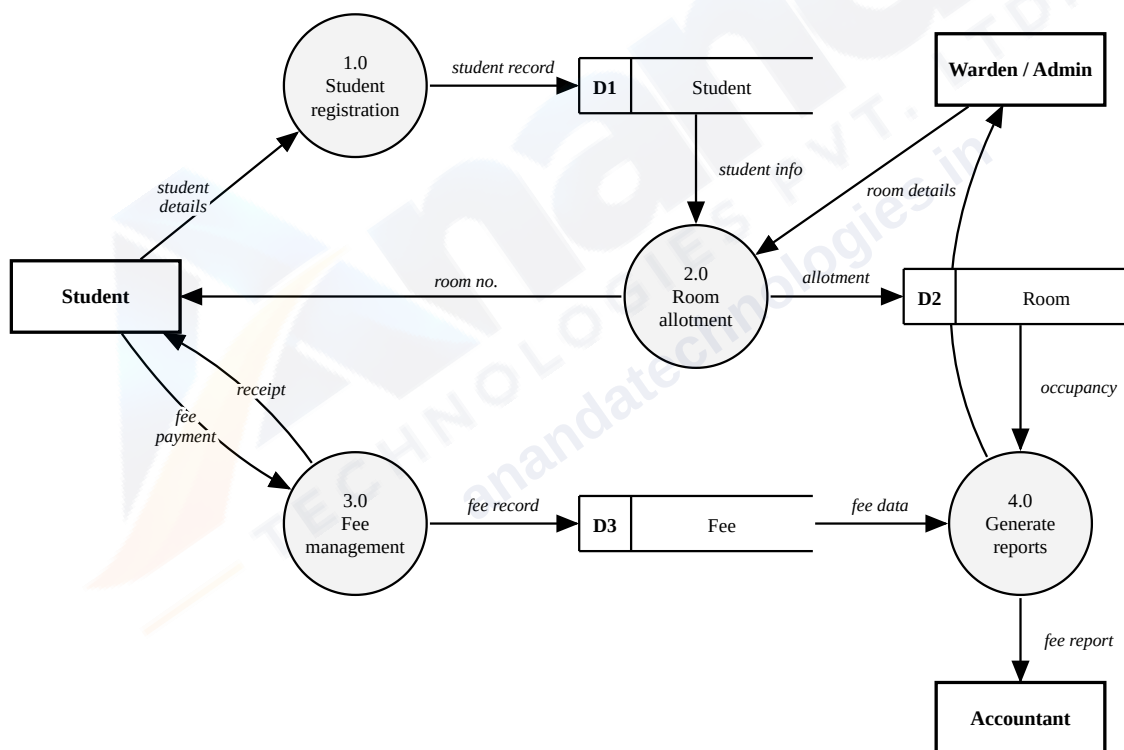


Fig.: Level 1 DFD — Hostel Management System

Bubbles = processes, rectangles = external entities, open-ended boxes = data stores, labelled arrows = data flows. Inputs and outputs of the context diagram balance with level 1.

(Level 0 DFD 2 M, level 1 DFD 2 M)

Q.3

a) Use case diagram — ATM system.

4 M

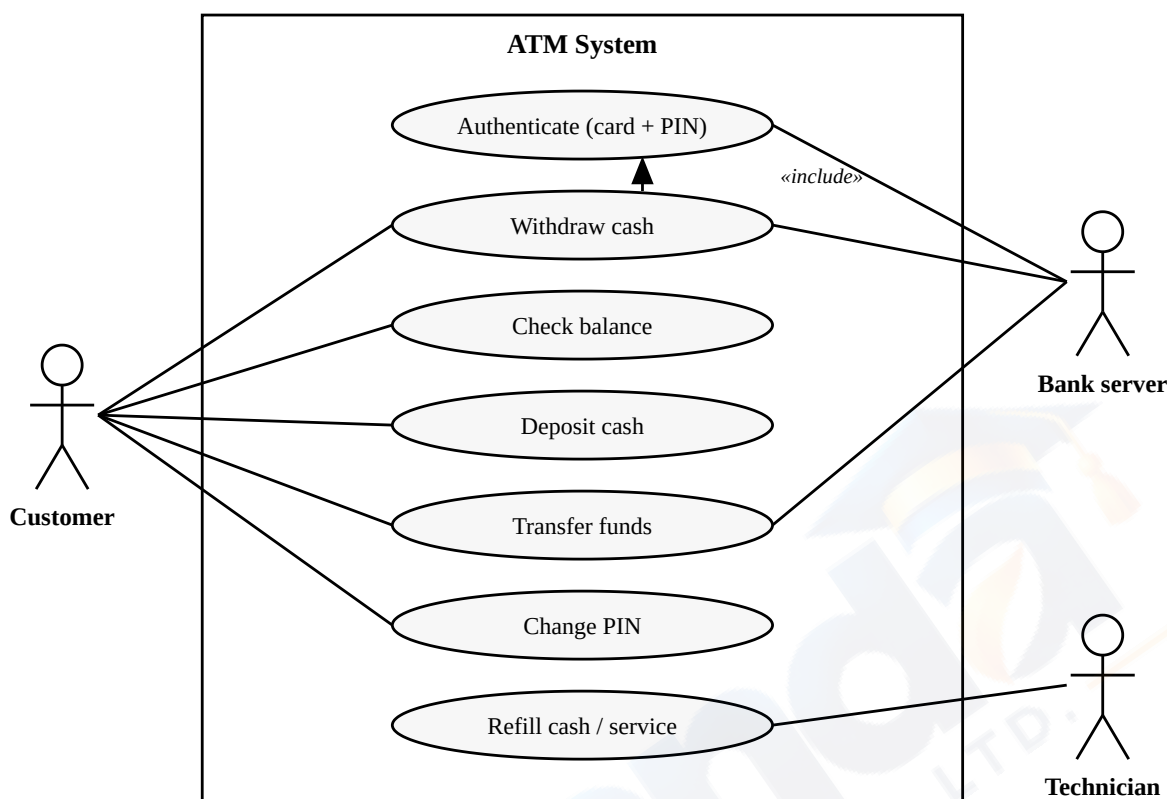


Fig.: Use case diagram — ATM System

- **Actors:** Customer (primary), Bank server (secondary system actor), Technician.
- **Withdraw cash «include» Authenticate** — every transaction first validates the card and PIN.

(Actors 1 M, use cases 2 M, relationships 1 M)

b) Static and dynamic testing.

4 M

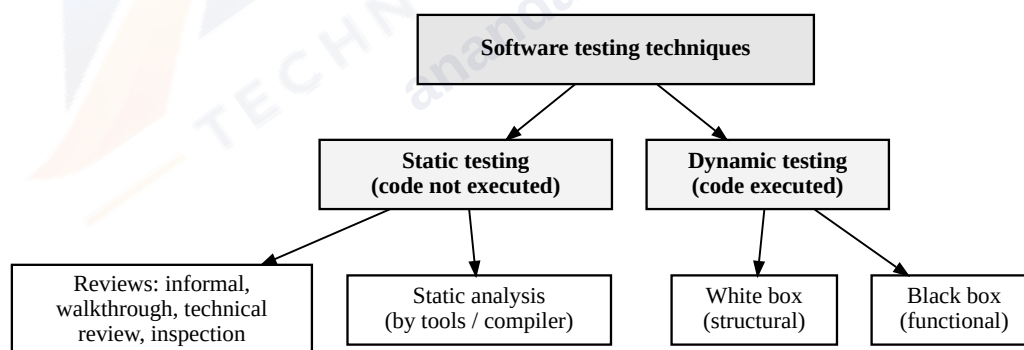


Fig.: Static and dynamic testing techniques

Static testing: the software work products (SRS, design, code) are examined *without executing* the code. Techniques: informal review, walkthrough, technical review, inspection and static analysis by tools. It finds defects early when they are cheap to fix.

Dynamic testing: the code is *executed* with test inputs and the actual output is compared with the expected output. Techniques: black-box (equivalence partitioning, boundary value analysis) and

white-box (statement, branch, path coverage); performed at unit, integration, system and acceptance levels.

Point	Static testing	Dynamic testing
Meaning	Testing without executing the code — the work products are examined	Testing by executing the code with test inputs and checking outputs
Stage	Early stage (requirements, design, code)	After code is compiled / build available
Techniques	Informal review, walkthrough, technical review, inspection, static analysis	Black-box (equivalence partitioning, BVA ...) and white-box (statement, branch, path)
V & V	Verification	Validation

(Static 2 M, dynamic 2 M)

c) Core principles of software engineering practice.

4 M

- 1. The reason it all exists:** A software system exists for one reason — to provide value to its users. Every decision should be checked against this.
- 2. KISS (Keep It Simple, Stupid!):** Design should be as simple as possible, but not simpler; simple designs are easier to understand and maintain.
- 3. Maintain the vision:** A clear vision (architecture) is essential; without it the system becomes a patchwork of incompatible designs.
- 4. What you produce, others will consume:** Always specify, design and code so that others (maintainers, testers, users) can understand your work.
- 5. Be open to the future:** Systems with a long lifetime must be designed to adapt to changes in requirements and platforms.

(Any four, 1 M each)

d) Test management and test infrastructure management.

4 M

Test management is the planning, organising, monitoring and controlling of all testing activities so that testing is completed on time, within budget and with the required quality. It covers test organisation, test planning, test design, execution, defect tracking and reporting, people management and integration with product release.

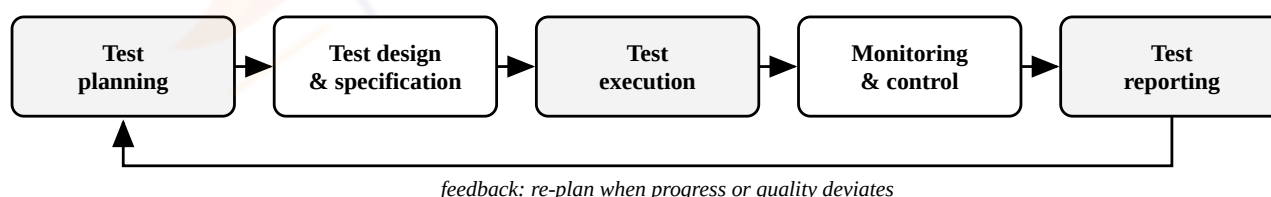


Fig.: Test management process

Test infrastructure management: testing needs a well-managed infrastructure made of three essential elements:

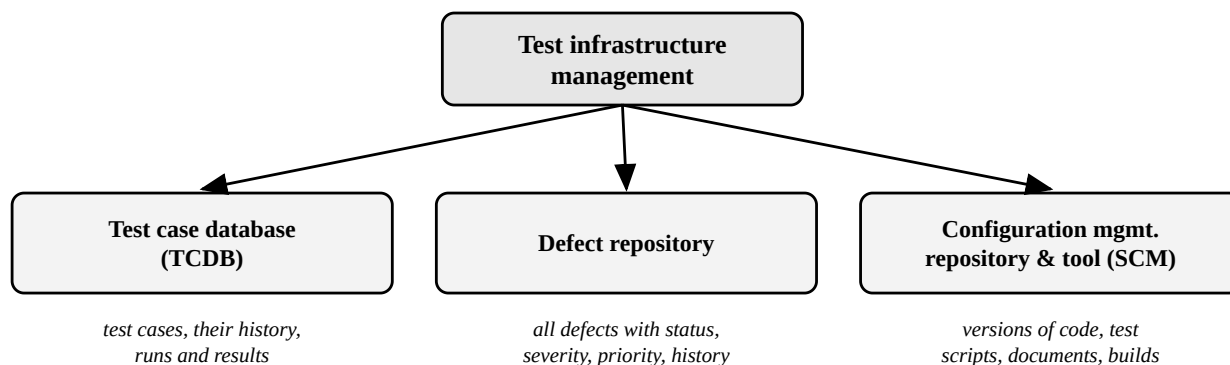


Fig.: Components of test infrastructure

1. **Test case database (TCDB):** stores all test cases, their relation to requirements and modules, and the history of every run and result.
2. **Defect repository:** stores details of all defects (status, severity, priority, history) — used for tracking and metrics.
3. **Configuration management repository and tool:** keeps track of versions of code, test scripts, documents and builds, so that the right version is tested and changes are controlled.

(Test management 2 M, test infrastructure 2 M)

Q.4

a) Size estimation techniques; LOC estimate.

4 M

Size estimation techniques: (1) Lines of Code (LOC), (2) Function Point (FP), (3) Feature point. LOC estimation by decomposition uses a three-point (beta) estimate for each function: $S = (S_{opt} + 4 S_m + S_{pess}) / 6$.

Function	Opt.	Most likely	Pess.	Expected LOC
User interface and login	1500	2000	3100	$(1500 + 4 \times 2000 + 3100) / 6 = 2100$
Question bank management	2200	3000	4400	$(2200 + 4 \times 3000 + 4400) / 6 = 3100$
Exam conduct and timer	2600	3400	5000	$(2600 + 4 \times 3400 + 5000) / 6 = 3533.3$
Result processing	1300	1800	2900	$(1300 + 4 \times 1800 + 2900) / 6 = 1900$
Reports and analytics	900	1400	2000	$(900 + 4 \times 1400 + 2000) / 6 = 1416.7$

Total estimated size = **12,050 LOC** $\approx$ 12.1 KLOC

Effort = size / productivity = 12,050 / 620 = **19.4 person-months**

(Techniques 1 M, expected LOC 2 M, effort 1 M)

b) CMMI levels.

4 M

CMMI (Capability Maturity Model Integration), developed by the SEI (Carnegie Mellon University), is a process meta-model that defines the characteristics an organisation's software process must have to be mature. In the **staged** representation it has five maturity levels:

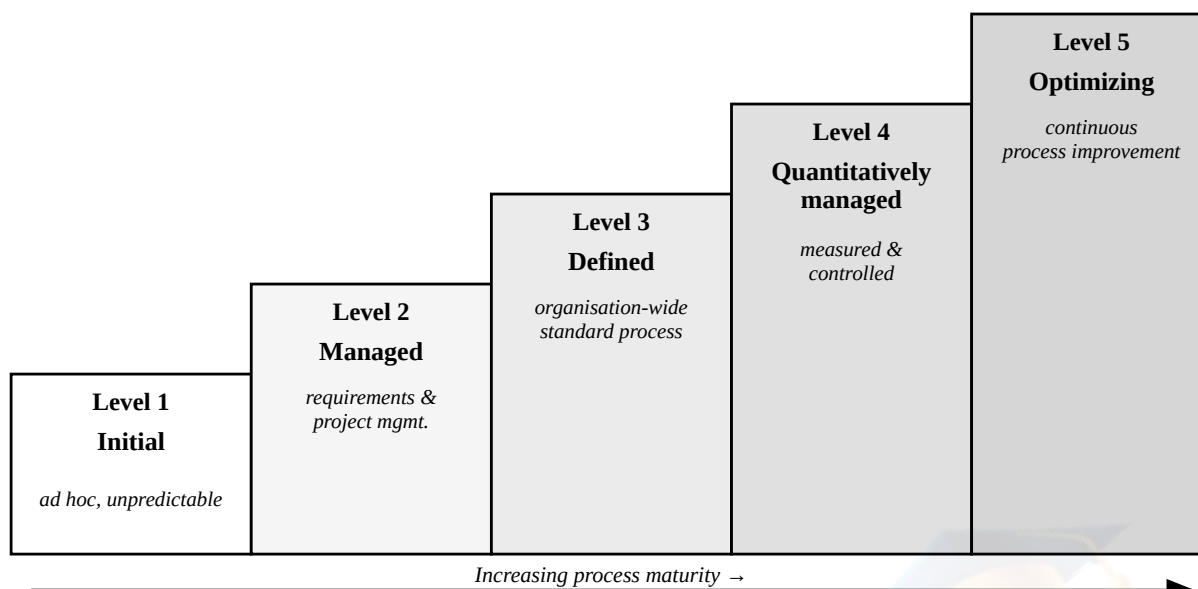


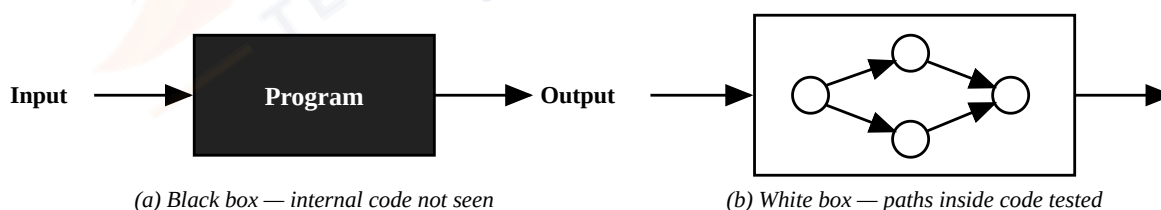
Fig.: CMMI maturity levels (staged representation)

1. **Level 1 — Initial:** Processes are ad hoc and chaotic; success depends on individual heroics. Projects often exceed budget and schedule.
2. **Level 2 — Managed:** Basic project management is established: requirements are managed, projects are planned, monitored and controlled; work products are under configuration management.
3. **Level 3 — Defined:** A standard, documented process is defined for the whole organisation and tailored for each project; risk management and verification/validation are practised.
4. **Level 4 — Quantitatively managed:** Detailed measures of process and product quality are collected; processes are controlled using statistical and quantitative techniques.
5. **Level 5 — Optimizing:** Focus on continuous process improvement through innovative ideas, technology and causal analysis of defects.

(Diagram 2 M, levels 2 M)

c) Black box vs white box testing.

4 M



Point	Black box testing	White box testing
Also called	Functional / behavioural / closed-box testing	Structural / glass-box / clear-box testing
Basis	Requirements (SRS) — internal code is not seen	Internal logic and code structure
Knowledge needed	No programming knowledge needed	Programming and implementation knowledge needed
Done by	Testers, end users	Developers / testers who know the code
Techniques	Equivalence partitioning, boundary value analysis, decision table, cause–effect graph	Statement, branch (decision), path coverage, basis path (cyclomatic complexity)
Levels	Mainly system and acceptance testing	Mainly unit and integration testing
Finds	Missing / incorrect functions, interface errors	Logic errors, unreachable code, wrong loops/conditions

(Any four points, 1 M each)

d) Need of metrics; product vs process metrics.

4 M

Measurement is the act of obtaining a measure — a quantitative indication of the extent, amount, dimension, capacity or size of some attribute of a product or process (e.g. 25 defects found). A **metric** is a quantitative measure of the degree to which a system, component or process possesses a given attribute (IEEE), usually relating measures, e.g. defects per KLOC.

Need:

1. To **understand** the product and process — what is happening during development and maintenance.
2. To **estimate** cost, effort and schedule of new projects from past data.
3. To **control** the project — track progress and quality against the plan (e.g. schedule variance).
4. To **assess quality** — defect density, reliability, test coverage tell whether the product is ready for release.

Point	Product metrics	Process metrics
Measures	Characteristics of the software product itself	Effectiveness / efficiency of the development and test process
Examples	LOC, function points, cyclomatic complexity, defect density, reliability (MTTF)	Defect removal efficiency, effort variance, productivity, defect arrival rate, test progress
Used for	Assessing quality and size of the product	Long-term process improvement
Collected	On work products at any stage	Across projects over a period of time

(Need 2 M, difference 2 M)

e) Test strategy vs test plan.

4 M

Point	Test strategy	Test plan
Meaning	High-level description of the testing approach — <i>how</i> testing will be carried out	Detailed document describing scope, objectives, schedule, resources and activities — <i>what, when, who, how much</i>
Level	Organisation / product level; may cover many projects	Specific to one project or release
Prepared by	Project manager / test manager (senior)	Test lead / test manager
Change	Static — rarely changes	Dynamic — updated as the project changes
Contents	Scope and objectives, test levels and types, techniques, tools, test environment, risks, release control	Test items, features to be tested / not tested, approach, pass/fail criteria, deliverables, schedule, staffing, responsibilities
Relation	Can be a section of the test plan	Derived from and implements the test strategy

(Any four points, 1 M each)

Q.5

a) Macro time-line chart — Online Examination System (15 days).

6 M

A **time-line chart** (Gantt-type chart) shows each work task as a horizontal bar against a calendar. Here the Online Examination System is planned for **15 working days** = 3 weeks × 5 days a week, with SDLC phases as the work tasks.

Work task (SDLC phase)	Start day	End day	Days	Week / day
Requirement gathering	1	2	2	W1 D1 – W1 D2
Planning & estimation	3	3	1	W1 D3 – W1 D3
Analysis (SRS)	4	5	2	W1 D4 – W1 D5
Design (DB, UI)	6	7	2	W2 D1 – W2 D2
Coding	8	11	4	W2 D3 – W3 D1
Testing	12	14	3	W3 D2 – W3 D4
Deployment & training	15	15	1	W3 D5 – W3 D5

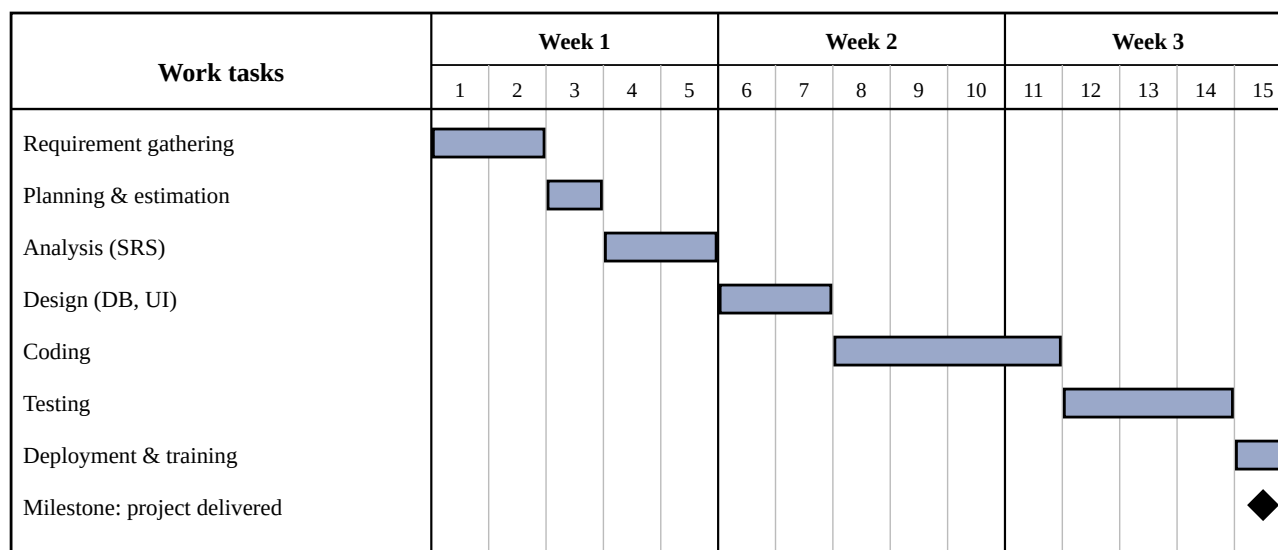


Fig.: Time-line chart — Online Examination System (15 working days, ◆ = milestone)

Communication with the college and the SRS are finished in week 1; coding and testing take most of weeks 2–3; the system is delivered on day 15 (milestone).

(Task allocation 2 M, chart with weeks/days 4 M)

b) Basic COCOMO — 120 KLOC, all three modes.

6 M

COCOMO (Constructive Cost Model) was proposed by Barry Boehm (1981). It is an empirical (heuristic) model that estimates effort, development time and staff from the size of the software in KLOC. It has three levels — **Basic**, **Intermediate** (adds 15 cost drivers) and **Detailed** (cost drivers per phase).

Project modes:

- **Organic:** small, simple project (about 2–50 KLOC), small experienced team, familiar environment, e.g. payroll, inventory.
- **Semi-detached:** medium size (about 50–300 KLOC), mixed team experience, e.g. compiler, DBMS.
- **Embedded:** large/complex (over 300 KLOC) with tight hardware, software and operational constraints, e.g. air-traffic control, ATM network.

Basic COCOMO equations: Effort $E = a \times (\text{KLOC})^b$ person-months · Development time $D = c \times (E)^d$ months · Staff = E / D persons

Mode	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.2	2.5	0.32

Given: Size = 120 KLOC = 120,000 LOC

Organic mode ($a = 2.4$, $b = 1.05$, $c = 2.5$, $d = 0.38$):

i) Effort $E = a \times (\text{KLOC})^b = 2.4 \times (120)^{1.05} = 2.4 \times 152.45 = 365.89$ person-months

ii) Development time $D = c \times (E)^d = 2.5 \times (365.89)^{0.38} = 23.55$ months

Semi-detached mode ($a = 3.0$, $b = 1.12$, $c = 2.5$, $d = 0.35$):

i) Effort $E = a \times (\text{KLOC})^b = 3.0 \times (120)^{1.12} = 3.0 \times 213.15 = \mathbf{639.45 \text{ person-months}}$

ii) Development time $D = c \times (E)^d = 2.5 \times (639.45)^{0.35} = \mathbf{23.99 \text{ months}}$

Embedded mode ($a = 3.6$, $b = 1.2$, $c = 2.5$, $d = 0.32$):

i) Effort $E = a \times (\text{KLOC})^b = 3.6 \times (120)^{1.2} = 3.6 \times 312.62 = \mathbf{1125.43 \text{ person-months}}$

ii) Development time $D = c \times (E)^d = 2.5 \times (1125.43)^{0.32} = \mathbf{23.68 \text{ months}}$

Mode	Effort (PM)	Time (months)
Organic	365.89	23.55
Semi-detached	639.45	23.99
Embedded	1125.43	23.68

Observation: embedded mode needs the most effort for the same size because of tight constraints.

(Formulae $1\frac{1}{2} M$, each mode $1\frac{1}{2} M$)

c) Requirements — Pharmacy Management System.

6 M

i) Medicine stock module

- Pharmacist shall add / update medicines with name, batch number, manufacturer, MRP, purchase price, quantity and expiry date.
- System shall reduce stock automatically after every sale and increase it after every purchase entry.
- System shall alert when stock falls below the re-order level.
- System shall list medicines expiring within the next 30 / 60 days.
- Search medicine by name, generic name or batch.

ii) Billing module

- Cashier shall create a bill by selecting medicines and quantity; price is taken from stock.
- System shall calculate GST, discount and net amount automatically.
- System shall not allow sale of expired medicines or quantity more than available stock.
- System shall print / SMS the bill and accept cash, card or UPI payment.
- Daily / monthly sales report shall be generated.

iii) Supplier module

- Admin shall add / edit supplier details — name, GST number, contact, medicines supplied.
- System shall create purchase orders for low-stock medicines.
- Purchase bills from suppliers shall be entered and linked to stock.
- System shall track payments due to each supplier.
- Supplier-wise purchase report shall be available.

Non-functional requirements (common to all modules):

- Only authorised users (admin, pharmacist, cashier) can log in; role-based access.
- Bill generation should take less than 2 seconds.
- Daily backup of the database.

Q.6

a) Translating the requirements model into the design model.

6 M

Each element of the requirements (analysis) model provides information that is necessary to create the four design models required for a complete design specification. The flow of information during software design is shown below:

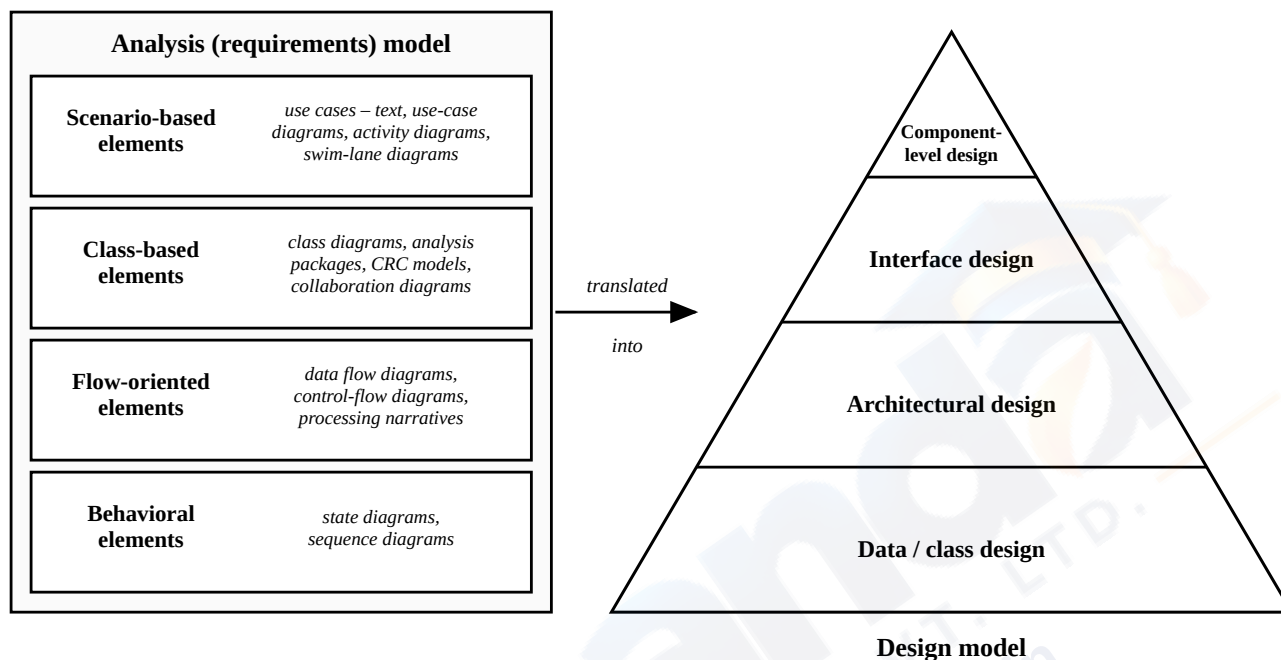


Fig.: Translating the requirements (analysis) model into the design model

1. **Data / class design** — transforms class models (class-based elements: class diagrams, CRC models) into design class realisations and the data structures required to implement the software.
2. **Architectural design** — defines the relationship between major structural elements, the architectural styles and patterns; derived from class-based and flow-oriented elements and the system specification.
3. **Interface design** — describes how the software communicates with systems that interoperate with it and with humans who use it; derived from scenario-based (use cases), flow-oriented and behavioral elements (state diagrams).
4. **Component-level design** — transforms structural elements of the architecture into a procedural description of software components; derived from class-based, flow-oriented and behavioral elements.

Design quality is assessed at each step; the design model must be traceable back to the requirements model.

(Diagram 2 M, explanation 4 M)

b) Basis path testing — grade().

In basis path testing (white box), a compound condition such as $m < 0 \parallel m > 100$ is drawn as two separate predicate nodes.

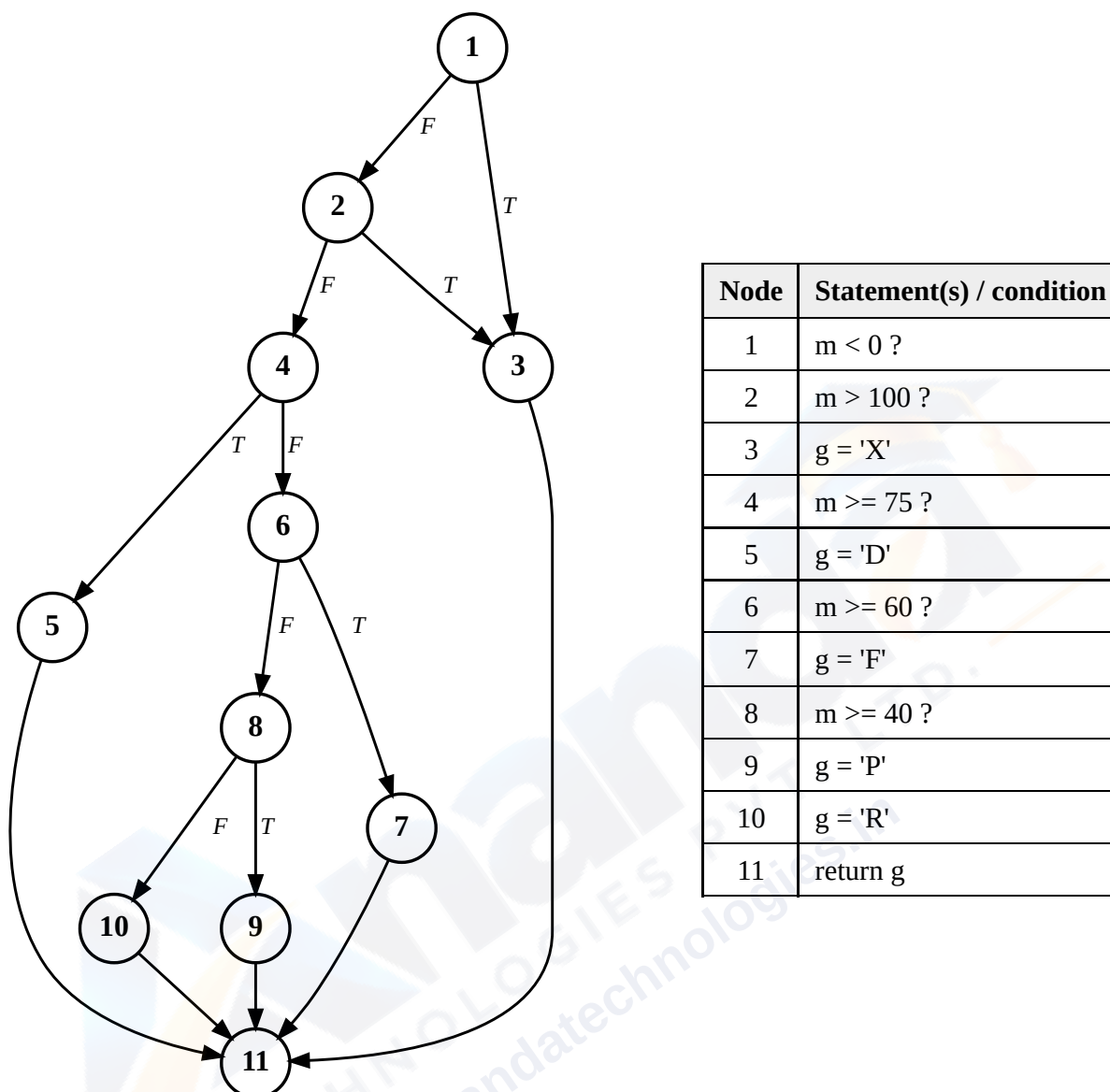


Fig.: Flow graph of grade() (compound condition split into nodes 1 and 2)

Cyclomatic complexity $V(G)$ (McCabe) gives the number of **independent paths** in the basis set, i.e. the minimum number of test cases that execute every statement and every branch at least once.

From the flow graph: number of nodes $N = 11$, number of edges $E = 15$, predicate (decision) nodes $P = 5$ (nodes 1, 2, 4, 6, 8).

i) $V(G) = E - N + 2 = 15 - 11 + 2 = 6$

ii) $V(G) = P + 1 = 5 + 1 = 6$

iii) $V(G) = \text{number of regions of the flow graph (including the outer region)} = 6$

Basis set of independent paths:

1. $1 \rightarrow 3 \rightarrow 11$

2. $1 \rightarrow 2 \rightarrow 3 \rightarrow 11$

3. $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 11$

4. $1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 11$
5. $1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 11$
6. $1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 10 \rightarrow 11$

Test cases (one per independent path; outputs obtained by compiling and running the function):

Test case	Path covered	Input m	Output grade
TC1	$1 \rightarrow 3 \rightarrow 11$	-5	X
TC2	$1 \rightarrow 2 \rightarrow 3 \rightarrow 11$	150	X
TC3	$1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 11$	82	D
TC4	$1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 11$	65	F
TC5	$1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 11$	48	P
TC6	$1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 10 \rightarrow 11$	20	R

(Flow graph 2 M, V(G) by three methods 2 M, paths and test cases 2 M)

c) Six Sigma — DMAIC and DMADV.

6 M

Six Sigma is a widely used strategy for statistical quality assurance, popularised by Motorola in the 1980s. It uses data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating defects. Six Sigma quality means not more than **3.4 defects per million occurrences**.

Six Sigma defines two methodologies, depending on whether an existing process is improved or a new one is designed.

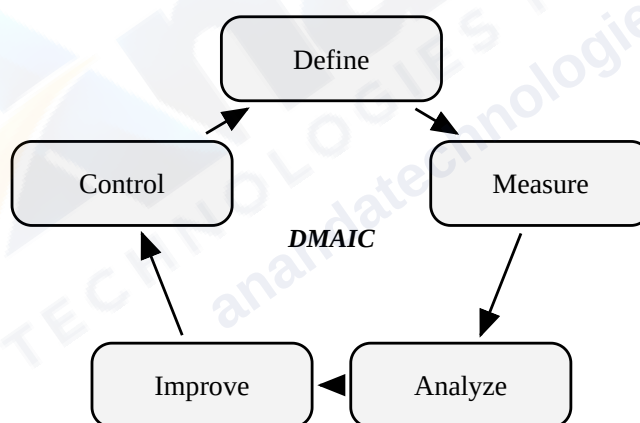


Fig.: DMAIC cycle (Six Sigma — improving an existing process)

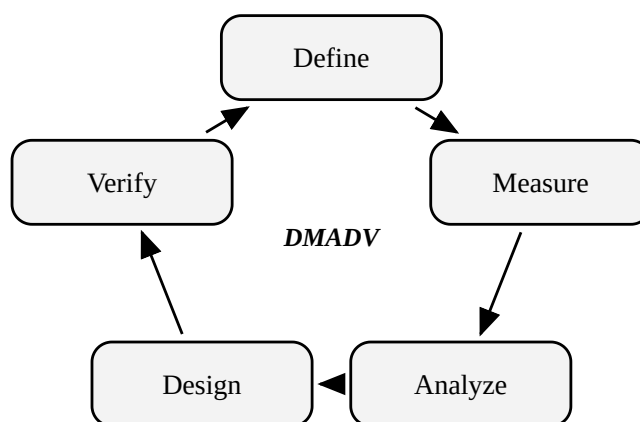


Fig.: DMADV cycle (Six Sigma — designing a new process)

DMAIC (improve an existing process that is below specification):

1. **Define** — customer requirements, deliverables and project goals through well-defined methods of customer communication.
2. **Measure** — the existing process and its output to determine current quality performance (collect defect metrics).
3. **Analyze** — defect metrics and determine the vital few causes.
4. **Improve** — the process by eliminating the root causes of defects.
5. **Control** — the process to ensure that future work does not reintroduce the causes of defects.

DMADV (develop a new process / product):

1. **Define** — customer requirements and project goals.
2. **Measure** — customer needs and specifications (critical-to-quality characteristics).
3. **Analyze** — process options to meet the customer requirements.
4. **Design** — the process (details) to meet the customer requirements.
5. **Verify** — the design performance and its ability to meet customer requirements.

(Six Sigma 1 M, DMAIC 2½ M, DMADV 2½ M)

Note: This is a **model question paper prepared by Ananda Technologies** only for practice and exam preparation, based on the pattern of previous MSBTE question papers and the K-scheme syllabus. It is **not an official MSBTE paper**, and it does **not** mean that these questions will come in the exam. Answers follow standard textbook points; marks shown are indicative. Free for students — anandatechnologies.in