ALLAMA IQBAL OPEN UNIVERSITY

USMAN NOTES
www.usmannotes.com

Name:

Registration No.:

Course Code: 3414

Program: PGD (CS)

Semester: Autumn 2026

ALLAMA IQBAL OPEN UNIVERSITY, ISLAMABAD

Department of Computer Science

Course: Software Engineering-I (3414) • Level: PGD (CS)

Semester: Autumn 2026

Assignment No. 2

Total Marks: 100 • Pass Marks: 50

SOLVED ASSIGNMENT GUIDE — For Study & Reference

USMAN NOTES www.usmannotes.com

Note: This guide is prepared for study and reference purposes. Students are advised to read the relevant units of the course book, understand the concepts, and then write the assignment in their own words. Do not copy this guide verbatim. The diagrams and the specification developed in this guide are original teaching examples designed for the AIOU systems named in the questions; adapt them in your own words rather than reproducing them exactly.

Course Overview

Assignment No. 2 moves from principles to the working products of software engineering. It requires a data flow diagram and an entity-relationship diagram for the Learning Management System of AIOU, an analysis of modular design judged by coupling and cohesion, an analysis of testing strategies together with a complete test plan for a login system, and a full software requirement specification for the Library Management System of AIOU. The diagrams in this guide were drawn for these questions and are explained component by component in the text; the test plan and the SRS follow the standard structures used in professional practice. As in Assignment No. 1, each answer is presented with an introduction, sub-headings, tables or

diagrams where they help, and a conclusion. One transparency note: Question 3(b) refers to "each software module scenario" without printing a separate list of scenarios, so the answer evaluates the standard module scenarios of the course text — the recognised coupling and cohesion types, each illustrated with a concrete module situation.

Question 1: DFD for the AIOU Learning Management System; What Went Wrong in Functional Modelling (20 Marks)

Question: (a) Design a DFD (Data Flow Diagram) for a Learning Management System of AIOU and explain its components. (b) The system fails due to poor modeling. Identify what went wrong in functional modeling

Introduction

A data flow diagram is the principal tool of functional modelling: it shows what a system does by tracing how data enters the system, is transformed by processes, is held in data stores, and leaves as outputs, without saying anything about how the transformations are physically implemented. For a learning management system (LMS) at Allama Iqbal Open University the fit is natural. AIOU serves very large numbers of distance learners, so its LMS must register students, deliver course material, collect and mark assignments, and keep students, tutors and administrators informed — all of them flows of data between identifiable actors. Part (a) designs a levelled DFD for such a system and explains each component and symbol. Part (b) then turns the same lens on failure: when a system built from a poor functional model collapses

in use, the defects can almost always be traced to specific, nameable modelling mistakes, and identifying them is the first step in teaching modellers not to repeat them.

Part (a): The System, Its Boundary and Its Actors

Before any diagram is drawn, the scope must be fixed. Our LMS covers student registration and login, enrolment in courses for a semester, delivery of study material and multimedia content, assignment submission, tutor marking and feedback, and reporting and notification. It does not itself process fee payments — that crosses the boundary to the bank — nor does it print and post physical books, though it records dispatch information supplied to it. The external entities, shown as squares, are therefore the Student, who registers, studies, submits and receives results; the Tutor/Teacher, who uploads material, sets and marks assignments and records grades; the Administrator, who creates courses, allocates tutors and manages accounts; and the Email/SMS Gateway, an external service through which alerts and results notifications leave the system. At the context level the whole LMS is a single process, System 0, exchanging data with these entities; the level-1 diagram below decomposes it into its four major processes. This levelling discipline — one clear top view, decomposed step by step, with the flows across each boundary preserved — is what keeps a DFD understandable.

USMAN NOTES www.usmannotes.com

Figure: Level-1 Data Flow Diagram — AIOU Learning Management System

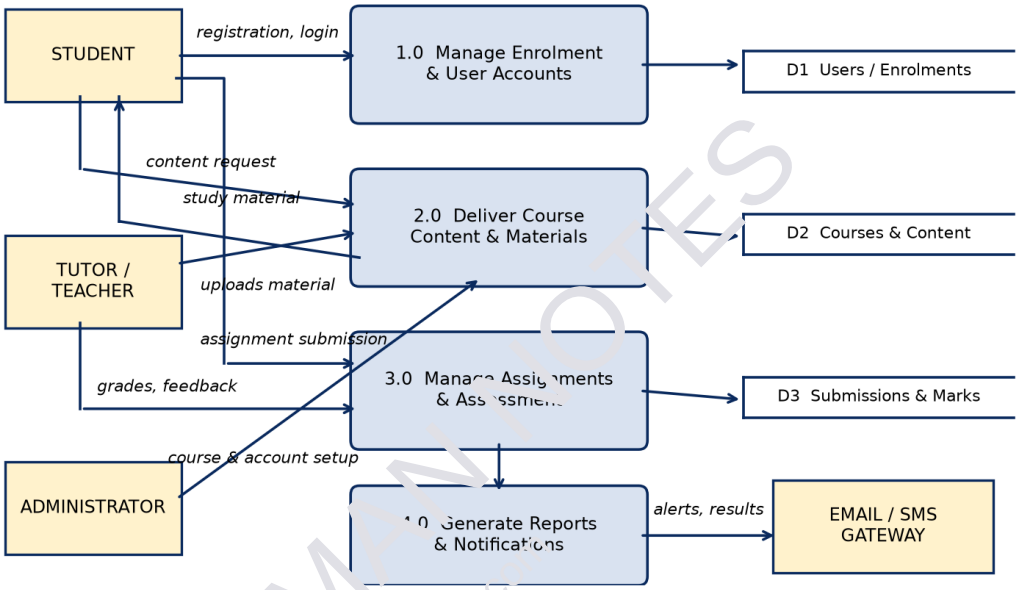


Figure 1: Level-1 DFD of the AIOU Learning Management System. Squares are external entities, rounded boxes are processes, arrows are data flows, and the open-ended bars D1–D3 are data stores.

Explaining the Components of the DFD

The diagram uses the four standard components of the notation. A process, drawn as a rounded box and numbered, transforms incoming data into outgoing data and is always named with a verb and an object. Process 1.0, Manage Enrolment and User Accounts, receives registration and login details from the student and account and course setup from the administrator, validates them, and writes accounts and enrolments to the data store; its outputs include confirmations back to the student. Process 2.0, Deliver Course Content and Materials, receives content requests from students and uploaded material from tutors, and returns study material — the recorded lectures, books in digital form, slides and announcements of the student's enrolled courses. Process 3.0, Manage Assignments and

Assessment, receives assignment submissions from students and grades and feedback from tutors, enforces due dates, and stores submissions and marks. Process 4.0, Generate Reports and Notifications, reads the results of the other processes and produces management reports for the administrator and alerts and result messages that leave through the email/SMS gateway.

A data flow, drawn as a labelled arrow, is data in motion, and its label names the data — "assignment submission", "grades, feedback", "study material" — never a vague word such as "information". A data store, drawn as an open-ended bar and numbered D1, D2, D3, is data at rest: D1 holds users and enrolments, D2 holds courses and their content, and D3 holds submissions and marks. Stores matter because they are where processes that run at different times communicate — the tutor's marking in process 3.0 can happen days after the student's submission precisely because D3 holds the work between them. An external entity, finally, is a person, organisation or other system outside the boundary that is a source or destination of data; entities never connect directly to each other or to a store, because data must always pass through a process that validates or transforms it. Three consistency rules complete the explanation: balancing, which requires that the flows entering and leaving a process at one level reappear when that process is decomposed at the next; the rule that every process must have both inputs and outputs, since a process with only inputs is a data black hole and one with only outputs creates data from nothing; and the rule that flows are named for what the data is, so that the data dictionary can define each one exactly.

Table 1: The four DFD components and their instances in the AIOU LMS

Component	Symbol	Instances in this system
External entity	Square	Student; Tutor/Teacher; Administrator; Email/SMS Gateway
Process	Rounded box, numbered	1.0 Enrolment & accounts; 2.0 Content delivery; 3.0 Assignments & assessment; 4.0 Reports & notifications
Data flow	Labelled arrow	Registration and login; content request; study material; assignment submission; grades and feedback; alerts and results
Data store	Open-ended bar, numbered D	D1 Users/Enrolments; D2 Courses & Content; D3 Submissions & Marks

USMAN NOTES www.usmannotes.com

Part (b): What Went Wrong in Functional Modelling When Such a System Fails

Now suppose the delivered LMS fails in use: students cannot see their courses, submissions vanish, tutors mark work that reappears as unmarked, and nobody trusts the reports. A

functional-modelling post-mortem typically finds the same family of errors. The first is an unbalanced or incomplete decomposition: the context diagram promised flows — say, a flow of tutor allocations from the administrator — that never appear at level 1, so the function was silently dropped and discovered missing only at delivery. The second is missing processes for exceptions. Novice models draw the happy path — a student submits, a tutor marks — and omit the flows that real life guarantees: late submissions, resubmissions after rejection, file-format failures, a tutor replaced mid-semester, a course withdrawn after enrolment. Because the model never contained them, the software never handled them, and the exceptions, which are a large share of real traffic in a university of AIOU's size, become failures.

The third error is defective data stores: flows drawn directly from an entity to a store, stores that are written but never read or read but never written, and the same data held in two stores without a defined rule for which is authoritative — the classic cause of a student marked present in one place and absent in another. The fourth is vague or missing data definitions: flows labelled "details" or "information" hide the fact that different stakeholders mean different things, so the enrolment process expects a fee-verification field that the admission process never supplies — a modelling gap that becomes an integration failure. The fifth is confusing the logical with the physical model: drawing the existing paper forms, offices and clerks into the DFD, so the new system faithfully reproduces the delays and duplications of the old manual process instead of modelling what the data must do. The sixth is scope failure at the boundary — modelling the bank's internal payment processing as part of the LMS, or forgetting the notification gateway entirely, so deadlines pass silently. Underlying several of these is a process error rather than a drawing error: the model was built by analysts alone, never walked

through with students, tutors and administrators using real scenarios, so its mistakes were protected from the very people who would have caught them. The remedies write themselves as the mirror image of the errors — balanced levelling, exception flows modelled deliberately, a maintained data dictionary, entity-to-store flows forbidden, logical modelling before physical choices, and structured walkthroughs of the model with users before design begins.

Conclusion

To conclude, the DFD in part (a) models the AIOU LMS as four numbered processes transforming labelled flows between three external actors and a notification gateway, resting on three data stores, and obeying the rules of balancing and complete input–output behaviour. Part (b) shows that when such a system fails through poor modelling, the causes are specific and preventable: unbalanced decomposition, omitted exception flows, defective stores and definitions, physical clutter in a logical model, a wrongly drawn boundary, and models never validated with their users. Functional modelling earns its place in the life cycle because every one of these errors costs a conversation to fix in the diagram and a crisis to fix in the delivered system.

Question 2: ERD for the AIOU Learning Management System, with Justification of Design Choices (20 Marks)

Question: Develop an ERD for the Learning Management System of AIOU and justify your design choices.

Introduction

If the data flow diagram answers the question "what does the system do?", the entity-relationship diagram answers the equally important question "what must the system remember, and how are those facts connected?" An ERD identifies the entities about which data is held, their attributes, and the relationships between them, together with the cardinality of each relationship — how many instances of one entity may be associated with an instance of another. This question asks for an ERD for the Learning Management System of AIOU and, just as importantly, for the justification of the design choices: an ERD is a set of decisions, and a designer who cannot explain a decision cannot defend the database that will be built from it. The design below keeps the scope of Question 1 — students, courses, tutors, enrolments, assignments and submissions — and every major choice is argued from the reality of how AIOU works.

USMAN NOTES www.usmannotes.com

Figure: Entity-Relationship Diagram — AIOU Learning Management System

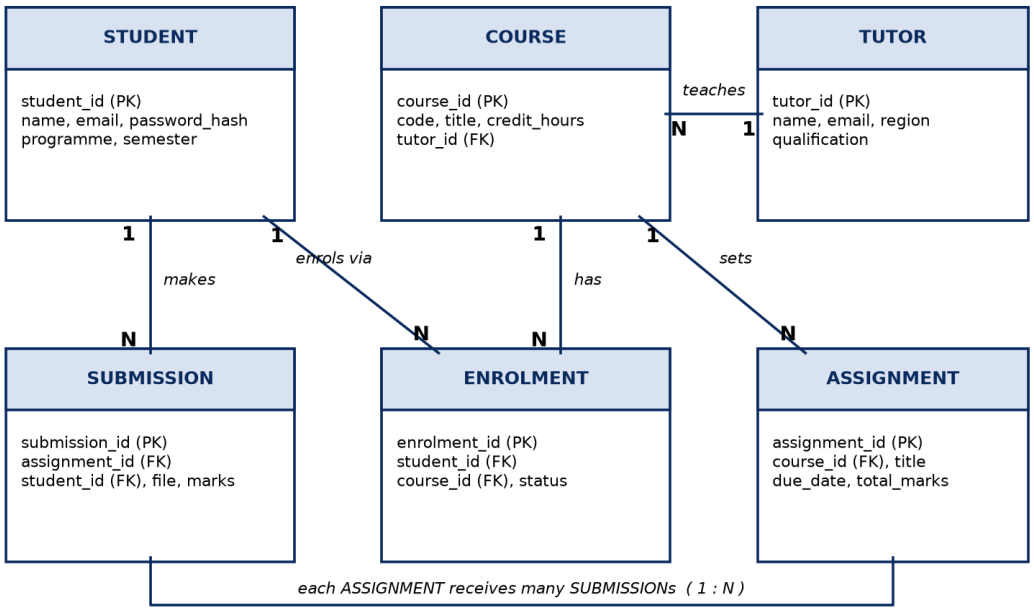


Figure 1: ERD for the AIOU LMS. Each box is an entity with its key attributes; lines are relationships labelled with their cardinalities. ENROLMENT is an associative entity resolving the many-to-many relationship between STUDENT and COURSE.

The Entities and Their Attributes

Six entities are identified. STUDENT holds the learners: student_id as primary key, with name, email, a securely hashed password (never the plain password — a design choice with direct security consequences), programme and current semester. COURSE holds what is taught: course_id as key, the course code and title as printed in the prospectus, and credit hours. TUTOR holds the teachers allocated to courses and regions: tutor_id, name, email, region and qualification. ENROLMENT is the pivotal entity: enrolment_id as its own key, student_id and course_id as foreign keys, plus enrolment date and status, because an enrolment is an event with a life of its own — it can be confirmed, suspended, completed or withdrawn — and not merely a line joining two tables. ASSIGNMENT holds the assessed tasks set for a course: assignment_id, course_id as foreign key, title, due date and total marks. SUBMISSION holds each student's response to an assignment: submission_id, assignment_id and student_id as foreign keys, the submitted file reference, submission time, marks awarded and feedback. Choosing these six, and no more, reflects the stated scope: fee payment, for example, is deliberately left outside this LMS model, as it was in the DFD, because it belongs to the finance and banking systems with which the LMS exchanges only a verification result.

The Relationships and Their Cardinalities

The relationships, each readable in both directions, are as follows. A TUTOR teaches many COURSEs, and each COURSE is taught by one tutor in a given offering, so TUTOR to COURSE is one-to-many, implemented by placing tutor_id as a foreign key in COURSE. A STUDENT makes many SUBMISSIONs and each SUBMISSION is made by one student: one-to-many. A COURSE sets many ASSIGNMENTs and each ASSIGNMENT belongs to one course: one-to-many, with course_id held in ASSIGNMENT. An ASSIGNMENT receives many SUBMISSIONs and each SUBMISSION answers one assignment: one-to-many. The remaining relationship is the instructive one. A STUDENT enrols in many COURSEs, and a COURSE is taken by many STUDENTs: a genuine many-to-many relationship. It cannot be implemented directly in a relational database, and it is not a bare link either, because facts belong to the pairing itself — the semester of the attempt, the enrolment status, and eventually the result are properties neither of the student alone nor of the course alone. ENROLMENT therefore becomes an associative entity in its own right, with its own primary key and one-to-many relationships to both STUDENT and COURSE, converting the many-to-many into two clean one-to-many relationships and giving the pairing's attributes a proper home.

Table 1: Relationships in the AIOU LMS ERD and how each is implemented

Relationship	Cardinality	Implementation
TUTOR teaches COURSE	1 : N	tutor_id foreign key in COURSE

STUDENT enrols in COURSE	M : N, resolved	ENROLMENT associative entity with student_id and course_id foreign keys
COURSE sets ASSIGNMENT	1 : N	course_id foreign key in ASSIGNMENT
ASSIGNMENT receives SUBMISSION	1 : N	assignment_id foreign key in SUBMISSION
STUDENT makes SUBMISSION	1 : N	student_id foreign key in SUBMISSION
ENROLMENT records an attempt	1 : N to both parents	ENROLMENT carries semester, status and result of one attempt

USMAN NOTES www.usmannotes.com

Justification of the Main Design Choices

Each major choice can now be justified. First, surrogate keys: every entity uses a system-generated identifier (student_id, course_id and so on) rather than a natural value such as email or course code as its primary key, because natural values change — students change email addresses, course codes are revised between editions of a prospectus — and a key that changes breaks every reference to it. The natural values are still held as attributes, with email and course code declared unique so duplicates are rejected. Second, the associative ENROLMENT

entity: as argued above, semester, status and result belong to the student-in-course pairing, and AIOU's reality makes this unavoidable, since the same student may attempt the same course again in a later semester after failure or deferral; only a separate ENROLMENT row per attempt can hold both attempts' histories truthfully. Third, storing marks and feedback on SUBMISSION rather than on ASSIGNMENT or STUDENT follows the same logic: marks are a fact about one student's one piece of work, and placing them anywhere else would force duplication or loss. Fourth, separating TUTOR from a general user table is justified at this level of modelling because tutors carry attributes — region and qualification — and relationships that students do not, though the design notes honestly that a production system would likely generalise STUDENT and TUTOR under a common USER entity for login purposes, with role-specific detail held separately; the ERD keeps them distinct because the question concerns the teaching relationships, and the trade-off is recorded rather than hidden.

Fifth, normalisation: the attributes are arranged so that no fact is stored twice. The course title lives only in COURSE, so correcting a title corrects it everywhere; the student's name lives only in STUDENT, so a submission displays it by reference instead of by copy, and the two can never disagree. This is the practical meaning of designing to third normal form, and its absence is exactly what produces the contradictory records discussed in Question 1. Sixth, integrity rules are declared with the diagram, not left to programmers' memory: a SUBMISSION cannot exist for an assignment in a course the student is not enrolled in; marks cannot exceed the assignment's total marks; a course cannot be deleted while enrolments exist. Finally, the model is deliberately kept at the conceptual level — entities, attributes, keys, relationships — without committing to a database product, column types or indexes, because those are design decisions

for the next stage and would obscure the business facts the ERD exists to agree with the university. The diagram was also validated against scenarios in the same spirit as the DFD: a student re-attempting a failed course, a tutor replaced mid-semester (the enrolment history survives because it references the course, and the course's tutor can be updated or, in a fuller model, versioned by semester), and a course offered in two programmes at once — each scenario maps onto the six entities without strain, which is the practical test of an ERD.

Conclusion

To conclude, the ERD models the AIOU LMS with six entities — STUDENT, COURSE, TUTOR, ENROLMENT, ASSIGNMENT and SUBMISSION — connected by five relationships whose cardinalities are stated and implemented through foreign keys. The design choices are each justified by the way the university actually works: surrogate keys because natural identifiers change, an associative ENROLMENT because attempts carry their own facts and repeat across semesters, marks placed on SUBMISSION because they belong to one student's one piece of work, and normalisation and integrity rules declared up front so that the database can never hold two contradictory versions of the same fact. An ERD defended in this way is more than a picture; it is the agreed memory of the system, on which the database, and ultimately the trustworthiness of every result the university publishes, will rest.

Question 3: Modular Design; Coupling and Cohesion Evaluated (20 Marks)

Question: (a) Poorly designed systems are difficult to maintain. Analyze the importance of modular design. (b) Evaluate coupling and cohesion in each software module scenario.

Introduction

Most software is not written once and discarded: it is maintained for years, and maintenance — correcting defects, adapting to new rules and adding features — consumes the majority of the total cost of most systems. Whether that maintenance is routine or ruinous is decided at design time, and the single most influential design decision is modularisation: how the system is divided into modules, what each module is responsible for, and how the modules are connected. Part (a) analyses the importance of modular design from the maintainer's point of view. Part (b) evaluates the two measures by which modular designs are judged — coupling, the strength of connection between modules, and cohesion, the strength of connection within a module. One point of transparency: the question paper speaks of "each software module scenario" but does not print a separate list of scenarios, so part (b) evaluates the standard set of module scenarios used in the course text — the recognised types of coupling and cohesion, each illustrated with a concrete module situation — which is the content the question is examining.

Part (a): Why Modular Design Decides Maintainability

A module is a well-bounded unit of software — a function, class, service or subsystem — with a defined interface through which other modules use it, and an implementation hidden behind that interface. Modular design matters to maintenance first because it localises change. In a system built as one tangled whole, a small correction — a changed tax rate, a revised deadline rule — requires the maintainer to understand, and risks breaking, the entire system. In a modular system the same change is confined to the module that owns that rule; its interface stays the same, so every other module is provably unaffected. This is the principle of information hiding: each module hides a design decision likely to change, so that when the decision changes, the change stops at the module boundary. The result is not merely convenience but safety, because the leading cause of maintenance disasters is the unintended side effect — a fix in one place silently breaking behaviour somewhere the maintainer never looked.

Modular design also makes the system understandable in pieces, which is the precondition of all maintenance. A maintainer can hold one module's responsibility in mind, test it in isolation with its own test cases, and replace or rewrite it without re-reading the whole system — an impossibility in a monolith of any size. Independent modules can be developed and maintained by different people in parallel, reused in other systems (a well-built authentication or reporting module serves project after project), and deployed or scaled separately in modern architectures. Modularity supports incremental growth as well: new features arrive as new modules or new versions of existing ones rather than as surgery on a living whole. The classic principles that produce these benefits are worth naming: decompose by function and by likely

change, not by the order in which code happens to run; keep each module's interface small, explicit and stable; and judge the decomposition by the two measures of part (b), because modularity is not automatically virtuous — a system cut into modules that are tightly coupled and internally incoherent has the costs of modularity without its benefits, and is harder to maintain than an honest monolith.

Part (b): Coupling — the Connections Between Modules

Coupling measures how strongly modules depend on one another; the design goal is loose coupling, so that a change in one module rarely forces changes in others. The recognised types form a spectrum from worst to best. In content coupling, one module reaches inside another — branching into its middle or modifying its internal data; maintenance is nearly impossible because the modules are effectively one module in disguise, and this must never be accepted. In common coupling, modules share global data; a change to the shared structure ripples unpredictably to every user of it, the classic cause of side-effect defects. External coupling ties modules to an imposed external format or device, acceptable only where the external standard genuinely cannot change. Control coupling passes flags that direct another module's internal logic, so the caller must know how the callee works — a leak of implementation across the boundary. Stamp coupling passes a whole data structure when only a few fields are needed, coupling the callee to the structure's full definition. Data coupling, by contrast, passes only the simple data actually needed, through an explicit interface: loose, explicit and the normal target of good design. Message coupling, in which modules communicate only by passing messages with no shared state at all, is the loosest form and the basis of service-oriented designs. Evaluating a scenario therefore means asking what exactly crosses the boundary: the more of

another module's internals a module must know, the worse the coupling and the more dangerous the maintenance.

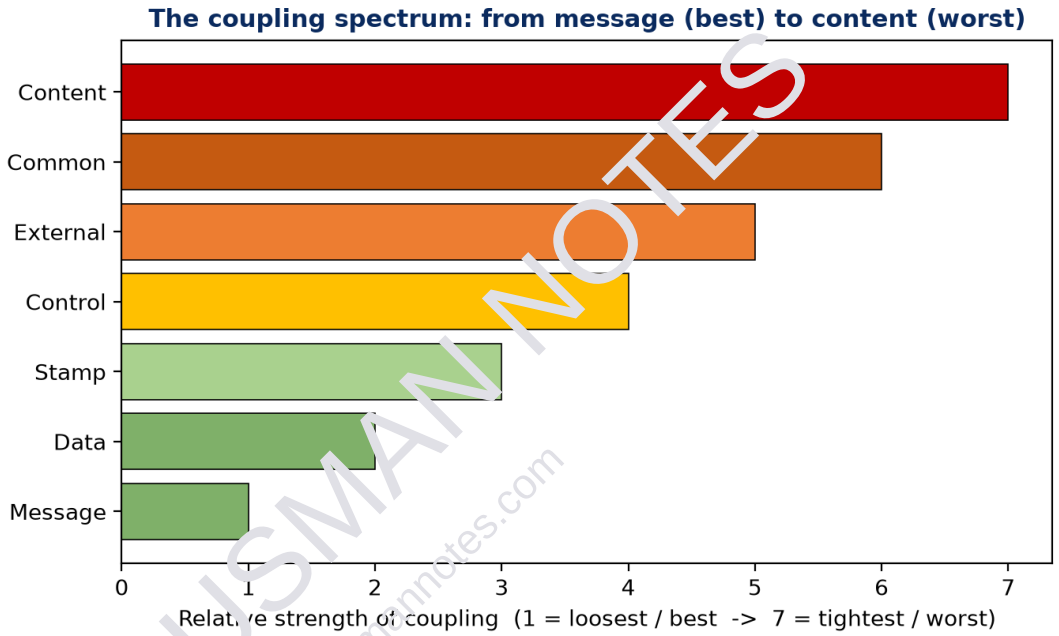


Figure 1: The coupling spectrum. Design should move modules toward the loose (data and message) end; content and common coupling are the usual root of maintenance failures.

Cohesion — the Unity Within a Module, and a Combined Evaluation

Cohesion measures how strongly the elements inside a module belong together; the goal is high cohesion, a module that does one thing completely. The scale, again from worst to best, runs as follows. Coincidental cohesion groups elements with no meaningful relationship — a "utilities" module of unrelated leftovers — and is unmaintainable because nothing predicts what it contains. Logical cohesion groups similar-looking operations selected by a flag, such as one module that reads from any device depending on a parameter, entangling unrelated code paths. Temporal cohesion groups elements that merely happen at the same time, like an

initialisation module that initialises everything. Procedural cohesion groups steps that follow a sequence but work on different data. Communicational cohesion groups operations that work on the same data, a respectable middle: all operations on a student record together. Sequential cohesion passes the output of one element as the input of the next, like a pipeline that validates, formats and stores a submission. Functional cohesion, the goal, means every element contributes to a single, well-defined task — a module that computes a student's aggregate result, and nothing else — making the module easy to name, easy to test and easy to reuse.

Table 1: Combined evaluation of representative module scenarios

Module scenario	Coupling	Cohesion	Verdict
A "miscellaneous utilities" module of unrelated helpers, used by everyone	Common (shared by all, changed by anyone)	Coincidental	Worst case: split it; give each helper a functional home
A login module that receives only user-id and password and returns a session token	Data / message	Functional	Model design: loose outside, unified inside; easy to test and replace
A report module passed the entire student record when	Stamp	Communicational	Acceptable but improve: pass only the fields needed,

it needs only name and marks			moving to data coupling
An initialisation module that sets up files, screens and printers at start-up	Data	Temporal	Tolerable at small scale; split by subsystem as the system grows
A grading module whose caller passes flags telling it which internal steps to run	Control	Procedural	Refactor: hide the steps; caller should state what it wants, not how
A module that directly modifies another module's internal marks array	Content	Cannot be judged — boundary already violated	Unacceptable: restore the interface; all access through the owning module

The combined lesson of the two measures is that they are checked together and can be traded against each other only within limits. Splitting a system into very many tiny modules can raise cohesion while multiplying coupling through endless calls; merging modules reduces coupling count while destroying cohesion. Skilled design searches for the natural joints of the problem — the entities and functions of the analysis model are usually those joints — where a module can be functionally cohesive inside and data-coupled outside. Evaluation in practice is done during

design reviews by exactly the questions used above: what crosses this boundary, and does everything inside this box serve one purpose? A module that passes both tests can be maintained by a stranger; a module that fails either will eventually be maintained by nobody, because nobody will dare.

Conclusion

To conclude, modular design is important because maintenance dominates the life of real software, and modularity — change localized behind stable interfaces, systems understandable and testable in pieces, work divisible among people and reusable across projects — is what makes maintenance safe and affordable. Coupling and cohesion are the instruments that tell a genuine modular design from a sliced-up monolith: design for loose coupling, ideally data or message coupling, and for high cohesion, ideally functional cohesion, and evaluate every module on both scales together. The poorly designed systems of the question are difficult to maintain for a precise reason that can now be stated in one sentence: in them, every change is global, because the modules neither hide their decisions nor mind their own business.

Question 4: Testing Strategies; A Test Plan for a Login System (20 Marks)

Question: (a) A software product fails after deployment. Analyze how testing strategies could prevent this. (b) Design a test plan for a login system, including different testing techniques.

Introduction

A product that fails after deployment fails at the most expensive and most public moment possible: real users are affected, data may be corrupted, emergency fixes are made under pressure, and the organisation's reputation absorbs damage that no later patch fully repairs. Yet most post-deployment failures are not bolts from the blue; they are defects that a suitable testing strategy would have exposed in private, at a fraction of the cost. Part (a) analyses how testing strategies, properly understood, prevent such failures. Part (b) makes the discussion concrete by designing a test plan for a small but genuinely critical system — a login system — showing the test items, techniques, cases, schedule logic and completion criteria that a plan must contain. Testing is used throughout in its proper sense: executing the software to reveal defects and to build justified confidence, recognising that testing can show the presence of defects but never prove their absence, which is why it must be combined with reviews and careful design.

Part (a): How a Testing Strategy Prevents Post-Deployment Failure

The first strategic principle is to test throughout the life cycle rather than at the end. In the V-model, every development stage has a corresponding test stage planned alongside it: requirements are the basis of acceptance testing, system design of system testing, detailed design of integration testing, and code of unit testing. Defects are therefore hunted at the level where they were introduced, while the people and documents that caused them are still at hand. A failed product typically shows the opposite pattern — testing compressed into the final weeks, the first activity sacrificed when the schedule slipped, so that whole categories of defect

reached the customer untested. The second principle is to test at multiple levels, because each level finds defects the others cannot. Unit testing by the developer catches logic errors in individual modules cheaply. Integration testing exposes the mismatches between modules — wrong assumptions about formats, sequences and error handling — that are the largest single source of serious failures in integrated systems. System testing examines the whole product against its specified requirements, functional and non-functional, in an environment resembling production. Acceptance testing puts the system in front of users and customers, whose real scenarios routinely reveal requirements misunderstandings that no internal test imagined.

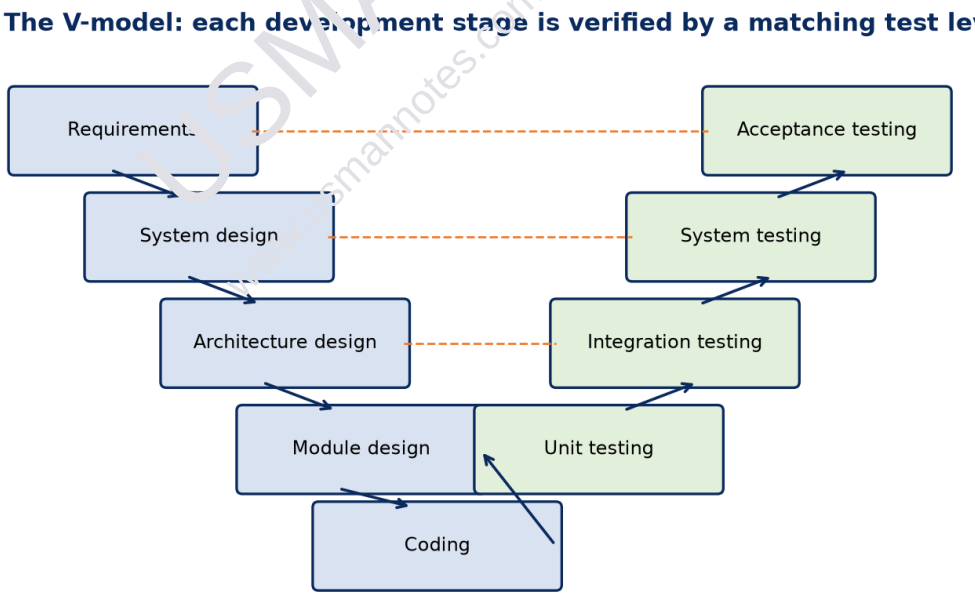


Figure 1: The V-model. Each development stage (left) is verified by a planned test level (right); failure after deployment usually means one or more of these levels was skipped or compressed.

The third principle is that a strategy names its techniques and its priorities in advance, guided by risk. Functional techniques — equivalence partitioning, boundary value analysis, decision tables and state-transition testing — cover the behaviour the specification describes; structural

techniques measure which code the tests have actually exercised; non-functional testing covers the qualities deployment failures so often involve: performance and load, because systems that pass with ten test users collapse with ten thousand real ones; security testing, because an untested login or input field is an open door; usability, compatibility and recovery testing, because users abandon or break systems that confuse them or lose their work when something fails. A risk-based strategy spends the deepest testing where failure would hurt most — payments, personal data, safety — instead of spreading effort evenly by habit. The fourth principle is process discipline around the tests themselves: a test plan written before testing begins, test cases traceable to requirements so that coverage can be audited, defects recorded, fixed and confirmed by retesting, and regression testing — re-running the accumulated test suite after every change — so that fixes stop reintroducing old failures, the most humiliating class of post-deployment defect. Finally, entry and exit criteria protect the release decision: testing may not start on an unstable build, and the product may not ship while specified criteria — all high-severity defects closed, planned coverage achieved — remain unmet. A strategy containing these elements does not guarantee perfection, but it converts deployment from a leap of faith into the last, best-evidenced step of a controlled process, and products protected by it fail in public rarely, mildly and recoverably.

Part (b): Test Plan for a Login System

The test plan below follows the standard structure of a test plan document (in the tradition of the IEEE test documentation standard) applied to the login function of a typical web application such as the AIOU LMS. Introduction and scope: the plan covers verification of user authentication — entering credentials, validation, session creation, password recovery, account

lockout and logout — for student, tutor and administrator roles. Test items: the login module, the credential store interface, the session manager and the password-reset function. Features to be tested include correct acceptance of valid credentials, rejection of invalid ones, lockout after repeated failures, reset by verified email link, session timeout and protection against common attacks; features not tested here are the internal correctness of the email service itself and the registration process, which has its own plan. The approach combines black-box functional techniques, white-box review of the authentication code, and non-functional testing for security and performance, executed first on the test environment with a prepared set of test accounts, and the plan states its suspension criteria (testing halts if login itself is broken on the delivered build, since nothing else can be reached) and its exit criteria (all planned cases executed, no open high-severity defects, and every requirement in the login section of the SRS traced to at least one passed case).

Table 1: Test cases for the login system, showing the technique each case applies

ID	Test case	Technique	Expected result
TC-01	Valid user-id and correct password	Equivalence partitioning (valid class)	Login succeeds; correct role dashboard opens; session created
TC-02	Valid user-id, wrong password	Equivalence partitioning (invalid class)	Login rejected with a general error; no hint which field was

			wrong
TC-03	Empty user-id and/or empty password submitted	Boundary / invalid input	Validation message; no request reaches the credential store
TC-04	Password of maximum allowed length; one character above it	Boundary value analysis	Maximum accepted; above-maximum handled as specified, never crashes
TC-05	Five consecutive wrong passwords (lockout limit = 5)	Boundary value + state transition	Account locks at the fifth failure; correct password also refused while locked
TC-06	Correct login immediately after lockout period expires	State transition testing	Login succeeds; failure counter reset
TC-07	SQL injection text entered in user-id field	Security / error guessing	Treated as literal text; login rejected; no database error shown

USMAN NOTES www.usmannotes.com

TC-08	Password reset requested for a registered email	Functional / scenario testing	Single-use reset link sent; old password stops working after reset
TC-09	Session left idle beyond the timeout limit	State transition / non-functional	Session expires; next action demands fresh login
TC-10	Five hundred simultaneous login attempts	Performance / load testing	Response within the specified limit; no successful logins lost
TC-11	Back button used after logout	State transition testing	Protected page not shown; user is returned to the login screen
TC-12	Tutor and administrator accounts log in	Role-based / decision table	Each role reaches only its own permitted functions

Three features of the plan deserve emphasis because students often omit them. First, the negative cases outnumber the positive ones, deliberately: a login system earns its keep by what it refuses, so wrong passwords, empty fields, injection attempts, locked accounts and expired

sessions are the heart of the plan, not afterthoughts. Second, the security expectations are written as testable results — the error message must not reveal whether the user-id or the password was wrong, passwords must never be stored or displayed in plain text, and credentials must travel only over an encrypted connection — because a requirement that cannot be tested cannot be verified. Third, the plan is traceable and repeatable: every case names its technique and expected result in advance, results are recorded case by case, defects are logged against case IDs, and the whole suite becomes the regression pack re-run whenever the authentication code changes. A plan of this shape fits on a few pages, yet it contains everything whose absence was lamented in part (a): scope, levels, techniques, criteria and evidence.

Conclusion

To conclude, post-deployment failure is usually the visible end of an invisible testing failure — levels skipped, techniques unnamed, non-functional qualities unexamined, release criteria waived — and a deliberate testing strategy prevents it by hunting defects at every level, in proportion to risk, with planned techniques and disciplined defect and regression management. The login test plan of part (b) shows what that means in miniature for one critical function: defined scope and items, twelve traceable cases applying equivalence partitioning, boundary value analysis, state-transition, security and load techniques, and explicit entry and exit criteria. Software will never be proven perfect by testing; it can, however, be shipped with evidence instead of hope, and that difference is the whole subject of this question.

Question 5: Software Requirement Specification for the Library Management System of AIOU (20 Marks)

Question: Develop a Software Requirement Specification for the Library Management System of AIOU.

Introduction

A software requirement specification (SRS) is the formal, agreed description of what a system must do and the qualities it must have, written so that the customer can validate it, the designer can build from it, and the tester can verify against it. This answer develops an SRS for the Library Management System of AIOU, following the classical structure of the IEEE 830 standard for such documents: introduction, overall description, and specific requirements covering functionality, external interfaces, performance, design constraints and quality attributes. The setting matters. AIOU's library system must serve a central library, regional and study-centre collections, and a vast distance-learning student body, many of whom will use the catalogue online rather than in person; it must manage books and other materials, members, issue and return, reservations, fines, and the acquisition of new stock. Requirements are stated in the testable style an SRS demands — each is specific, numbered and written so that its satisfaction can be checked — and the document closes with the supporting information a real specification carries.

1. Introduction and 2. Overall Description

1.1 Purpose: this SRS defines the requirements for the AIOU Library Management System (LMS-Lib) for use by the developers, the library administration and the university's auditors of the project. 1.2 Scope: the system shall computerise the catalogue, circulation, membership, reservation, fine and acquisition functions of the university's libraries; it shall not replace the university's finance system, to which it shall pass fine and payment summaries, nor the student information system, from which it shall obtain student membership data. 1.3 Definitions: a "copy" is one physical instance of a "title"; a "member" is a registered student, tutor or staff member entitled to borrow: the "circulation desk" is any authorised issue/return point. 1.4 References: the university library rules, the SRS standard structure, and the AIOU LMS requirements with which this system must interoperate. In the overall description, the product perspective places the system as one component of the university's information systems, exchanging membership data with the student system and fine summaries with finance. The main product functions are catalogue searching, member management, issue and return, reservation, fine calculation, stock acquisition and reporting. User classes are the librarian/administrator, who manages stock, members and rules; the circulation staff, who issue, return and collect fines; and members, who search the catalogue, reserve titles and view their own borrowing records. Operating environment: a web-based system usable from library counters and from members' own devices over the internet. The key design constraints are that the system must protect members' personal and borrowing data, must work acceptably on the low-bandwidth connections many distance learners have, and must preserve a complete audit

trail of circulation transactions. The central assumption is that every copy carries a unique accession number, applied as a barcode, by which it is identified at issue and return.

3. Specific Functional Requirements

The functional requirements are grouped by module and numbered for traceability. Catalogue: FR-1, the system shall allow any user, including guests, to search the catalogue by title, author, subject and ISBN, returning matching titles with their availability status within the performance limit of section 4. FR-2, the system shall display for each title the number of copies, their location by library or centre, and the expected return date of issued copies. Membership: FR-3, the system shall register members by importing verified student and staff records and shall issue each member a unique membership identity. FR-4, the system shall enforce borrowing limits by member class — for example, the number of items and the loan period set by library rules for students, tutors and staff — and shall block borrowing by members whose membership has expired or been suspended. Circulation: FR-5, the system shall record the issue of a copy by accession number to a member, computing the due date from the loan rules. FR-6, the system shall record returns, update availability immediately, and compute any overdue fine at the rate in force. FR-7, the system shall allow a loan to be renewed once, provided the title has not been reserved by another member. Reservation: FR-8, a member shall be able to reserve an issued title; FR-9, on its return the system shall hold the copy for the first reserving member and notify that member, releasing the hold after the period set by the rules. Fines and payments: FR-10, the system shall maintain a fine ledger per member and shall prevent further borrowing while unpaid fines exceed the limit set by library rules. Acquisition and stock: FR-11, library staff shall be able to add titles and copies, record their source and cost, and withdraw

damaged or lost copies with a reason recorded. FR-12, the system shall record a copy reported lost and raise the replacement charge prescribed by the rules. Reports: FR-13, the system shall produce, on demand, lists of overdue items, fine summaries for finance, stock and issue statistics per library, and the borrowing history of a member to authorised staff only.

Table 1: Selected functional requirements with their verification method

ID	Requirement (summary)	Priority	Verified by
FR-1	Catalogue search by title, author, subject, ISBN with availability	High	System test against a loaded catalogue
FR-5 / FR-6	Issue and return by accession number with due date and fine computation	High	Scenario tests including overdue and lost cases
FR-8 / FR-9	Reservation queue with hold and notification on return	Medium	Multi-member scenario test
FR-10	Fine ledger with borrowing block above the set limit	High	Boundary tests at and above the limit

FR-13	Overdue, fine and statistics reports; borrowing history restricted	Medium	Report inspection and access-control test

Interface, Performance and Quality Requirements

External interface requirements: the user interface shall be a browser-based interface in clear language, usable by counter staff with minimal training and by members on mobile phones; all circulation functions shall be operable by barcode scanner and keyboard. Hardware interfaces are limited to barcode scanners and receipt printers at counters. Software interfaces: membership verification against the student information system, fine summaries to the finance system in the agreed format, and email/SMS notification through the university's gateway. Communications shall use encrypted connections for all member data. Performance requirements: catalogue search shall return results within three seconds for ninety-five per cent of searches over a catalogue of the size the university holds; issue and return transactions shall complete within two seconds at the counter; the system shall support the peak load of the start-of-semester period, specified as concurrent users, without breaching these limits. Quality attributes: availability during library hours with recovery from failure within a stated time and no loss of committed circulation transactions; reliability such that no issue or return, once confirmed to the user, can be lost; security through role-based access — members see only their own records, circulation staff cannot alter loan rules, and only the librarian role can

withdraw stock or waive fines, with every such action written to the audit trail; usability measured by a new counter assistant completing an issue transaction after the standard short training; and maintainability in the precise sense that loan periods, fine rates and borrowing limits shall be changeable by authorised staff as configuration, without program modification — because library rules change far more often than library software should.

Supporting Information and a Note on Use

A complete SRS closes with supporting material: an appendix containing the data dictionary of the principal items (title, copy, member, loan, reservation, fine), the business rules in force at the time of writing listed separately so that rule changes can be traced, and an index of requirements. Two properties of the document deserve final emphasis for the student using this guide. First, every requirement above is stated so that it can be verified — by test, inspection, analysis or demonstration — and vague wishes such as "the system shall be fast and user-friendly" have deliberately been replaced by measurable statements; an SRS whose requirements cannot be checked is a source of disputes, not of software. Second, the SRS is a baseline under change control: when library rules or university needs change, the change is proposed, its impact on design, cost and schedule assessed, and the document amended formally, so that the specification always describes the system that is actually being built and used. Students should also note what this SRS deliberately leaves out — database structures, screen layouts and program design — because those are the products of the design stage that follows, and mixing them into the SRS would freeze implementation choices before the requirements themselves are agreed.

Conclusion

To conclude, this SRS specifies the AIOU Library Management System in the standard form: an introduction fixing purpose, scope and definitions; an overall description of the product's perspective, functions, users, constraints and assumptions; specific, numbered and verifiable functional requirements for catalogue, membership, circulation, reservation, fines, acquisition and reporting; and explicit interface, performance, security, availability and maintainability requirements, with loan rules held as configuration because they change most often. Written this way, the document can be validated by the library before anything is built, designed from without ambiguity, and tested against requirement by requirement — which is exactly the service a software requirement specification exists to perform, and the reason it stands at the head of every disciplined development project.

USMAN NOTES www.usmannotes.com