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. 1

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. Case studies mentioned are widely reported examples from the software engineering literature, summarised here for learning; students should present the analysis in their own words.

Course Overview

Software Engineering-I (3414) for PGD (CS) introduces the disciplined development of software: why engineering principles matter, how process models such as prototyping and RAD organise the work, how projects are managed, how systems are engineered as wholes of interacting subsystems, and how analysis models a problem before design begins. Assignment No. 1 applies these ideas to realistic situations — a startup destroyed by poor quality, a famous requirements failure, a client who cannot state what he wants, a delayed project, a hospital system and an online shopping platform, and the analysis of a university management system. Each answer in this guide follows the same structure — an introduction, developed sub-

sections with tables and diagrams where the question calls for them, and a conclusion — so that the student can understand the argument fully and then write the assignment in his or her own words.

Question 1: Software Quality, Engineering Principles, and a Requirements Failure Case (20 Marks)

Question: (a) A startup fails due to poor software quality. Analyze how software engineering principles could have prevented this failure. (b) Discuss a case where a lack of understanding of software needs and requirements led to system failure.

Introduction

A startup rarely fails because its idea was worthless; very often it fails because the software that carried the idea could not be trusted. Customers forgive a plain interface, but they do not forgive an application that loses their data, charges them twice, crashes at the moment of payment, or is so slow and confusing that a competitor is only one tap away. Part (a) of this question therefore asks us to analyse a familiar story — a young company destroyed by poor software quality — and to show, principle by principle, how software engineering could have prevented that failure. Part (b) asks for a concrete case in which the deeper cause was not careless coding but a failure to understand what the software was actually needed to do. The two parts belong together: quality is not something added at the end by testing; it is built in from the first conversation about requirements, and most expensive failures are requirements failures that were discovered far too late.

Part (a): How Poor Quality Destroys a Startup

Consider a typical composite case, of the kind reported again and again in post-mortems of failed technology startups. A small team builds a promising online service — say a food-delivery or booking application — in a few months. The first version works in demonstrations, early adopters sign up, and an investor funds expansion. Then success itself becomes the enemy. User numbers grow faster than the system was designed for, so response times collapse at peak hours. There was no version control discipline, so two developers overwrite each other's work and a fix for one bug quietly reintroduces another. Nobody wrote the requirements down, so every feature is remembered differently by the founder, the developer and the marketing team, and the product drifts. Testing consisted of the developers clicking through the happy path on their own machines, so the first real testers were paying customers. A database change made without a migration plan corrupts order records; payments are duplicated; refunds are handled by apology. Within a few months the rating of the application falls, churn rises, support costs eat the remaining cash, and a competitor with a merely adequate but reliable product takes the market. The startup did not run out of ideas; it ran out of trust, and trust in a software business is simply quality experienced by the user.

The Principles That Were Missing — and What Each Would Have Changed

The first missing principle is that software development must follow a defined process rather than improvisation. Software engineering replaces heroic, memory-based work with a process — requirements, design, implementation, verification and evolution — in which each activity has inputs, outputs and checks. Even a very small team using a lightweight process gains the

essentials: work is planned, decisions are recorded, and the state of the product is visible. Closely connected is the principle of rigorous requirements engineering. Had the startup elicited, analysed, specified and validated its requirements, it would have known its real quality requirements in advance: how many simultaneous users the system must support, what response time is acceptable, what happens when a payment fails midway, and which data must never be lost. Most quality disasters are non-functional requirements — performance, reliability, security, usability — that were never stated, and what is never stated is never designed or tested.

The second group of principles concerns design before code and modularity. A system designed as well-defined modules with clear interfaces can grow feature by feature; a tangled system in which every part depends on every other part can only be changed by gambling. Information hiding and abstraction mean that a change inside one module, such as replacing the payment gateway, does not ripple unpredictably through the whole product. Design reviews, even informal ones, catch structural mistakes when they cost hours instead of months. The third group is verification and validation practised continuously: unit testing of individual components, integration testing of their combinations, system testing against the requirements, and acceptance testing with real users. The well-known cost curve of defects, illustrated below, shows why this matters so much to a poor startup: a defect that costs one unit of effort to fix when it is a wrong sentence in the requirements can cost ten in design, more in code, and up to a hundred times more after release, when it also carries refunds, compensation and reputational damage.

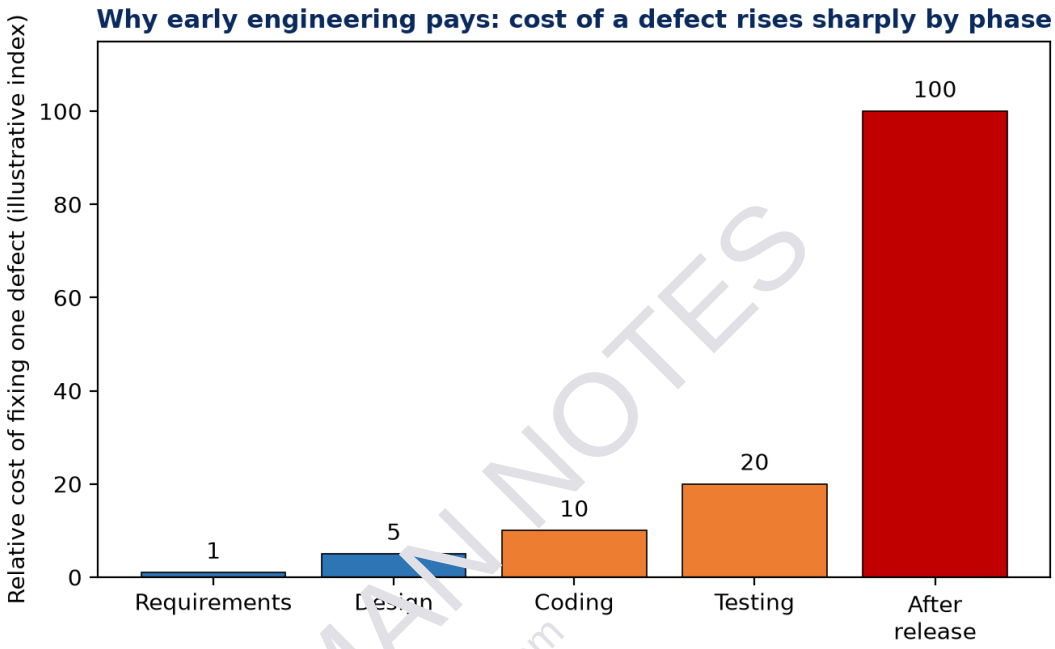


Figure 1: The relative cost of fixing a defect rises steeply the later it is found (illustrative index, after the classic software engineering cost studies). Finding problems early is the cheapest quality strategy a startup has.

The fourth group is configuration management and disciplined change. Version control, a proper branching and release practice, automated builds and a change-control habit — every change requested, assessed for impact, approved and recorded — would have stopped developers overwriting each other and would have made it possible to roll back a bad release in minutes instead of apologising for days. The fifth principle is project discipline: realistic estimation, a schedule with visible milestones, and risk management that names the top risks — scalability, a single overloaded developer, dependence on one external service — and prepares responses before they strike. Finally, software engineering insists that quality is a management concern, not only a technical one: defined quality goals, reviews and inspections, coding standards and a culture in which reporting a defect early is rewarded rather than

punished. A startup that works this way is not slower; it is faster in the only race that matters, because it does not keep rebuilding what it built badly.

Table 1: Failure symptom in the startup, the missing principle, and the preventive practice

Observed failure	Missing engineering principle	Practice that would have prevented it
System collapses under growing user load	Non-functional requirements never specified	State load and response-time requirements; load testing before launch
Fixes reintroduce old bugs; developers overwrite work	No configuration management	Version control, branching rules, automated build and regression tests
Product drifts away from what customers need	No requirements engineering or validation	Written, reviewed SRS; prototypes and user feedback before building
Customers become the first real testers	No systematic verification and validation	Unit, integration, system and acceptance testing with test plans
One database change corrupts live records	No design discipline or change control	Impact analysis, data migration plan, backup and rollback procedure

USMAN NOTES www.usmannotes.com

Deadlines missed, risks strike unprepared	No project/risk management	Estimation, milestones, risk register with responses

Part (b): A Case of Failure Through Misunderstood Requirements — Denver International Airport Baggage System

A classic, thoroughly documented case of requirements misunderstanding is the automated baggage-handling system built for Denver International Airport in the United States in the early 1990s. The airport was designed around a revolutionary, fully automated baggage system: thousands of computer-controlled carts on a vast network of tracks would carry every bag from check-in to aircraft without human handling. The vision was attractive, but the requirements were never properly understood by any of the parties. The airlines, whose operations the system had to serve, were brought in late and their detailed needs — sizes and weights of bags, transfer times, handling of oversized items such as skis, and what should happen when part of the system stopped — were not elicited systematically at the start. The airport authority and the contractor proceeded on a shared but shallow understanding: the system was treated as a construction and machinery problem rather than as one of the largest real-time software systems attempted up to that time.

As development proceeded, the consequences of that shallow understanding accumulated. Requirements kept changing as airlines finally examined the design and objected; each change to the physical layout demanded changes to control software that was already written. The complexity of the software — coordinating thousands of carts, sensors and junctions in real

time, and recovering gracefully from jams and faults — had been grossly underestimated precisely because the operational requirements had never been modelled in detail. Testing, compressed because the airport's opening date was fixed and public, revealed the system destroying bags, sending carts crashing into each other and misrouting luggage on a large scale. The airport's opening was delayed by well over a year at enormous daily cost, the automated system was drastically scaled back, a conventional manual baggage system had to be built as a backup, and the automated system was eventually abandoned by its principal airline user. Every element of the failure traces back to requirements: stakeholders identified late, needs assumed rather than elicited, no validated specification, uncontrolled change, and complexity discovered during testing instead of during analysis.

The lessons generalise far beyond airports. First, the people who will live with the system — operators, not only sponsors — must be involved from the beginning, because needs live in operational detail. Second, requirements must be modelled and validated, through prototypes, simulations and walkthroughs, especially where the system is novel; a requirement that has never been tested against reality is a guess. Third, feasibility must be assessed honestly: a requirement that demands unproven technology at unprecedented scale is itself a risk that analysis should expose. Fourth, change is inevitable, so it must be managed rather than absorbed silently into an already fragile design. The Denver case is taught in software engineering courses, alongside cases such as the Ariane 5 rocket loss — where software reused from an earlier rocket embodied assumptions about flight data that nobody re-examined for the new vehicle — precisely because both show the same moral: systems rarely fail because

programmers cannot code; they fail because somebody built the wrong system, confidently and at great expense.

Conclusion

To conclude, the startup of part (a) and the airport of part (b) failed for the same underlying reason: engineering discipline was treated as a luxury to be afforded later, when in software it is the only thing that makes "later" arrive. Defined process, rigorous requirements, modular design, continuous verification, configuration and change management, and honest project and risk management are not bureaucratic decorations; each one answers a specific, repeatedly observed way in which software projects die. A startup that practises even the lightweight core of these principles finds its defects when they are cheap, learns what its customers need before it builds the wrong thing, and earns the trust on which its survival depends. Software engineering, in short, is how good ideas are prevented from being killed by bad software.

Question 2: The Prototyping Model; RAD Compared with the Linear Sequential Model (20 Marks)

Question: (a) A client is unclear about requirements. Analyze how the prototyping model can help in this situation. (b) Compare RAD and Linear Sequential Model in the context of a fast-paced startup project.

Introduction

Requirements are the foundation of every software project, yet they are often the least solid part of it. Clients frequently know their problem only in a general way: they can describe their

business, their frustrations and their hopes, but they cannot state precisely what the software should do, and they often recognise what they want only when they see something they do not want. Building a complete system from such uncertain beginnings, and discovering the misunderstanding at delivery, is the most expensive pattern in software development. Part (a) analyses how the prototyping model answers exactly this situation. Part (b) then compares two contrasting process models — rapid application development (RAD) and the linear sequential, or waterfall, model — in the demanding context of a fast-paced startup, where speed matters but so does not building the wrong product.

Part (a): Why Unclear Requirements Are Dangerous

Unclear requirements are dangerous for three reasons. First, ambiguity multiplies: a vague sentence in a requirements discussion becomes several different interpretations in design, code and testing, and the interpretations rarely match the client's unspoken expectation. Second, the cost of correction rises steeply with time, as the defect-cost curve in Question 1 showed; a misunderstanding corrected in a conversation costs almost nothing, while the same misunderstanding corrected after release can require redesign of the database, the interfaces and the user's own business procedures. Third, unclear requirements cannot be tested against: acceptance becomes a matter of opinion and disappointment rather than a check against agreed behaviour. The traditional response — demanding that the client simply specify everything precisely in writing before work begins — fails because many clients genuinely cannot do so. They lack a mental model of what software can do, they cannot foresee detailed behaviour, and their own business understanding evolves as they think about the system. A process model is needed that turns this human limitation into part of the method itself.

How the Prototyping Model Works

The prototyping model answers unclear requirements by building, quickly and cheaply, a working model of the proposed system — or of its uncertain parts — and putting it in front of the client for reaction. In Pressman's classic description the paradigm begins with communication, in which developer and client define the overall objectives and identify whatever requirements are already known, outlining the areas where further definition is mandatory. A quick plan and quick design follow, concentrating on what the user will see: input formats, screens, navigation, outputs and reports. The prototype is then constructed, deployed, and evaluated by the client and users, whose feedback is used to refine the requirements. The cycle — listen, build or revise the mock-up, let the customer test-drive it, and listen again — repeats until the requirements are understood well enough to engineer the real product, as the figure shows.

USMAN NOTES www.usmannotes.com

The prototyping paradigm: build, show, listen, revise

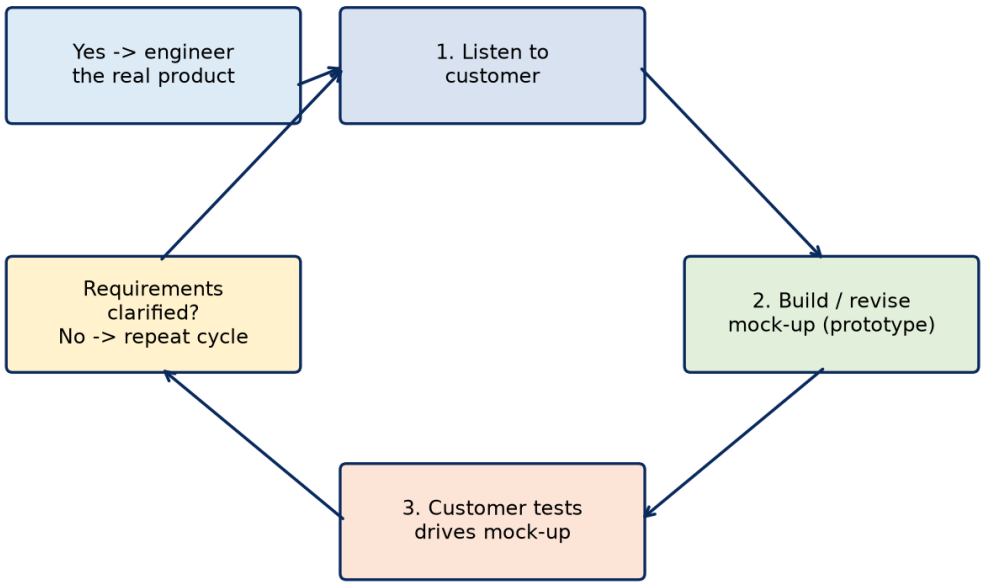


Figure 1: The prototyping paradigm. Each loop converts vague impressions into tested, specific requirements, because the client reacts to something real instead of imagining an abstract description.

The prototype helps in several distinct ways. It makes the abstract concrete: instead of asking "is this requirement correct?", which a client can only guess at, the developer asks "is this screen, this report, this sequence of steps correct?", which the client can answer from direct experience. It surfaces hidden requirements — the missing report, the exceptional case, the approval step that everybody forgot — because working through a realistic scenario reminds users of their real work. It exposes misunderstandings early, while correction means changing a mock-up rather than a production system. It gives the client and developer a shared vocabulary, since both now point to the same screens and behaviour when they discuss a feature. It also helps technically: a prototype can test the feasibility of a risky interface, a response-time target or an unfamiliar technology before the project commits to it. Finally, it builds the client's confidence and engagement, because progress is visible and the client becomes a partner in defining the product rather than a judge waiting at the end.

Throwaway and Evolutionary Prototyping; Cautions

Two forms of the model must be distinguished. In throwaway (rapid) prototyping, the prototype is built only to understand the requirements and is then discarded; the real system is engineered properly afterwards, using sound design and quality practices. In evolutionary prototyping, the prototype is built robustly and is refined step by step into the final system. Each form has its proper use, and confusing them is the model's main danger. The classic risks of prototyping must be managed honestly. The client who sees a working-looking prototype may believe the system is nearly finished and press for its immediate delivery, not realising that

the prototype omits error handling, security, performance work and most internal quality. The developer, under the same pressure, may make the prototype's quick compromises permanent, delivering a poorly structured system that is hard to maintain. Prototyping can also become endless if neither side defines when the requirements are clear enough to proceed. Good practice therefore agrees the purpose of each prototype in advance, states explicitly that the throwaway prototype will be discarded, records the refined requirements that emerge from each cycle in proper specification documents, and applies full engineering discipline to the real product. Used in this way, prototyping is not the opposite of engineering rigour; it is the front end of it, existing precisely for the situation in this question, where the client is willing but unclear.

USMAN NOTES www.usmannotes.com

Part (b): RAD and the Linear Sequential Model for a Fast-Paced Startup

The linear sequential model — the waterfall — arranges development as a one-way sequence of phases: communication and requirements, planning, modelling and design, construction, testing, and deployment with support. Each phase is completed and documented before the next begins, and the model assumes that requirements can be fully and correctly established at the start. Its strengths are order, visibility and control: progress is easy to measure, documentation suits contractual and regulated settings, and each phase has clear quality gates. Its weakness, decisive in a startup, is rigidity. Real projects rarely flow so calmly; requirements discovered to be wrong during testing can only be corrected by expensive return journeys against the current of the model, and the customer sees working software only very late.

Rapid application development is an incremental, speed-oriented adaptation of the sequential approach. The project is divided into small, functionally meaningful increments, each developed by a small team working in tight cycles of business modelling, data modelling, process modelling, application generation and testing, typically in periods measured in weeks. RAD leans heavily on reusable components, code-generating and visual development tools, and continuous, empowered user involvement, and it assumes the business can be modularised so that increments can proceed in parallel. For a startup, RAD's attractions are obvious: a usable product reaches the market in a fraction of the waterfall time, feedback arrives while the company can still change direction, and limited funds are spent first on the features users touch. Its costs are equally real: it demands skilled, committed teams and constantly available users; it performs badly when the system cannot be cleanly modularised, when performance or interoperability requirements are severe, or when technical risks are high and unexplored; and its speed can degenerate into undisciplined hacking if the modelling phases are skipped in the rush to generate screens.

Table 1: RAD and the Linear Sequential (Waterfall) Model compared for a startup project

Dimension	Linear Sequential (Waterfall)	RAD
Requirements assumption	Complete, correct and stable before design begins	Evolve through increments and constant user feedback
Time to first working product	Late — after the full sequence of phases	Early — first increment within weeks

Response to change	Costly and discouraged; change travels back through phases	Expected; absorbed by the next increment
User involvement	Heavy at start and acceptance, light in between	Continuous and empowered throughout
Team and tooling demands	Works with average teams; documentation driven	Needs skilled teams, reusable components and generation tools
Risk profile	Requirements risk discovered late, when it is dearest	Speed risk: poor modularisation or skipped modelling harms quality
Fit for a fast-paced startup	Poor, unless the product is well understood and regulated	Strong, provided the system modularises and users stay involved

USMAN NOTES www.usmannotes.com

For the fast-paced startup of this question, the analysis therefore favours RAD in most realistic conditions. A startup's scarcest resources are time and information: it does not yet know exactly what the market wants, and it cannot afford to spend a year discovering that it built the wrong product beautifully. RAD converts development into a sequence of experiments, each producing working software and market evidence. Yet the wise startup borrows waterfall virtues rather than rejecting them entirely: it still writes down the requirements of each

increment, designs before it generates, tests each increment properly, and keeps the documentation that maintenance, hiring and due diligence will later demand. Where a part of the product is stable, well understood and safety- or compliance-critical — a payment or records core, for example — a more sequential treatment of that component remains sensible inside an overall rapid, incremental strategy. Process models are tools, and the engineering skill lies in matching the tool to the risk.

Conclusion

To conclude, prototyping helps the unclear client because it replaces imagination with experience: each cycle of building, showing and listening turns vague wishes into specific, validated requirements while change is still cheap, provided the prototype's purpose is agreed, its lessons are recorded, and the final product is engineered rather than improvised. Between the two models of part (b), the linear sequential model offers order at the price of late feedback and rigidity, while RAD offers speed, early working software and continuous feedback at the price of demanding teams, users and modular design. For a fast-paced startup that must learn quickly, RAD — practised with engineering discipline — is normally the better choice, with sequential rigour reserved for the stable, critical core of the system.

USMAN NOTES www.usmannotes.com

Question 3: Project Management, Delay, and Management Problems in Large Systems (20 Marks)

Question: (a) A software project is delayed due to poor planning. Analyze the role of project management in avoiding such issues. (b) Identify and analyze common software management problems in large-scale systems.

Introduction

Ask experienced developers why software projects are late and the answers are seldom about typing speed. Projects are late because the work was underestimated, because the wrong people were assigned or arrived too late, because requirements grew while the deadline stood still, because risks that everybody sensed were never written down or answered, and because nobody knew the project was in trouble until the trouble was too large to hide. In other words, projects are late mainly for management reasons, which is why software project management is a distinct discipline within software engineering rather than a clerical extra. Part (a) analyses the role of project management in preventing the delay described in the question. Part (b) identifies and analyses the management problems that recur in large-scale systems, where size itself multiplies every difficulty.

Part (a): What Planning Failure Looks Like — and What Management Does About It

Poor planning in software usually has a recognisable anatomy. The estimate is produced before the requirements are understood, often by working backwards from a date somebody has already promised. The work breakdown is missing, so whole categories of work — data

migration, documentation, user training, integration with other systems, the correction of defects found in testing — are silently absent from the plan. Dependencies are ignored, so the plan assumes tasks can proceed in parallel that in reality must wait for one another. Buffers are zero, so the first surprise consumes the delivery date. Progress is then measured by opinion — "about ninety per cent done" for months — until integration and testing, the phases least compressible, reveal the true position. Project management exists to replace every element of this anatomy with a disciplined counterpart.

Its first role is estimation grounded in evidence. A project manager decomposes the product into a work breakdown structure, estimates size and effort for each element using historical data, expert judgement and, where possible, more than one technique, and converts effort into a schedule that respects dependencies and identifies the critical path — the chain of tasks whose delay delays the whole project. The figure of the constraint triangle summarises the manager's balancing act: scope, time and cost are linked, with quality at the centre, so a promise to add scope without moving time or cost is not optimism but arithmetic error. The manager's duty is to make that trade-off visible to decision-makers before they make it, not after. Closely allied is staffing and organisation: choosing the team structure, assigning clear responsibilities, recognising that adding people to a late project usually makes it later — because newcomers consume the time of the very people who must finish the work, a truth known as Brooks' Law — and protecting the team from constant interruption and task-switching.

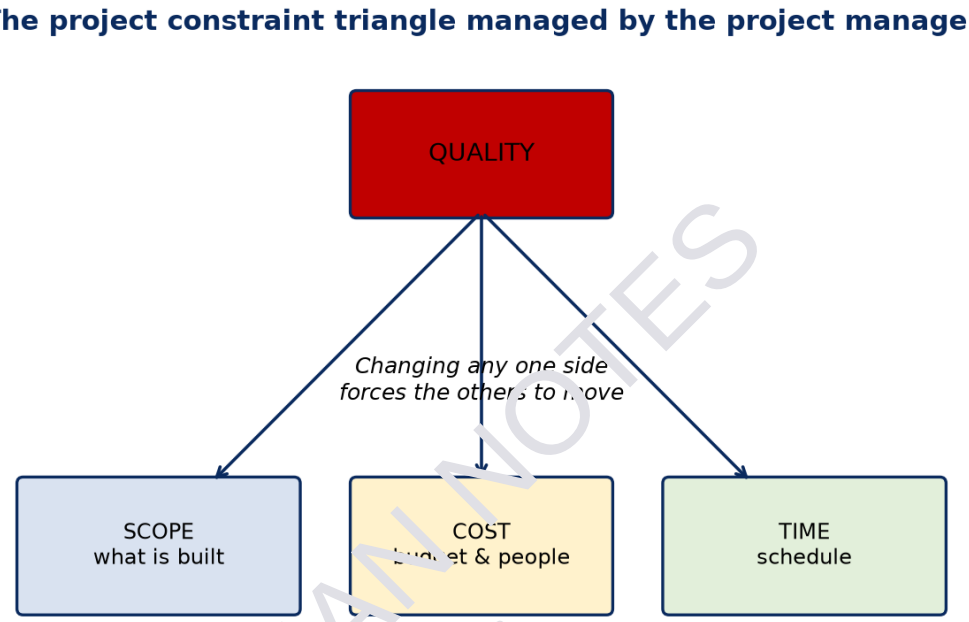


Figure 1: The project constraint triangle. Scope, time and cost cannot be changed independently; the project manager's role is to negotiate the balance openly instead of letting delay reveal it.

The second role is tracking and control. A plan prevents delay only if reality is continuously compared with it: milestones with objective completion criteria, earned-value or simple burn-down measures, regular reviews, and honest status reporting that separates "work started" from "work finished and tested". When variance appears — and the manager's assumption is that it will — control means acting early while options remain: re-sequencing work, reducing or phasing scope with the client's agreement, adding resources to genuinely parallel tasks, or renegotiating the date with evidence in hand. The third role is risk management: identifying risks before they become problems, assessing their probability and impact, and preparing responses — avoidance, mitigation, contingency plans — for the top risks, then monitoring them throughout the project. The fourth is communication and stakeholder management,

because many delays are really decision delays: a client who takes three weeks to approve a requirement has delayed the project exactly as surely as a slow programmer. Finally, the manager manages change and configuration, so that scope creep — the slow, unrecorded growth of requirements that is the single most common cause of software delay — is priced, approved and reflected in the plan, or else consciously refused. In the language of the discipline, project management addresses the four Ps: people, product, process and project — and delay is what happens when any one of them is left to chance.

Part (b): Common Management Problems in Large-Scale Systems

Large-scale systems — many users, many teams, many subsystems, long lives, integration with other systems — intensify every ordinary problem and add some of their own. The first is communication overhead. The number of communication paths in a team grows roughly with the square of its size, so coordination, not coding, becomes the dominant cost; decisions slow down, misunderstandings multiply across sites and time zones, and the architecture itself begins to mirror the communication structure of the organisation, whether or not that structure suits the system. The second is requirements complexity and volatility at scale: thousands of requirements from many stakeholder groups, some contradictory, changing over a development period measured in years, so that parts of the system are built against needs that have already moved. Managing this requires requirements traceability, prioritisation and a change process with genuine authority — all of which large projects often possess on paper and bypass in practice.

The third problem is estimation and planning error amplified by size. Large systems contain novel elements whose difficulty cannot be known in advance, integration work that is habitually underestimated, and external dependencies — vendors, other projects, regulators — outside the manager's control, so schedules built as single confident dates are almost guarantees of public delay. The fourth is architectural and integration risk: subsystems developed by different teams embody different assumptions, and the mismatch is discovered at integration, the most expensive moment, unless interface control, continuous integration and incremental delivery are enforced from the start. The fifth is quality management at scale: defect numbers grow with size, testing a large system exhaustively is impossible, environments and test data become projects in themselves, and without disciplined defect tracking and root-cause analysis the same classes of error recur for years.

USMAN NOTES www.usmannotes.com

Table 1: Common management problems in large-scale systems and their countermeasures

Problem	Why scale makes it worse	Management countermeasure
Communication overhead	Communication paths grow roughly as the square of team size	Clear team structure, defined interfaces, decision records, liaison roles
Requirements volatility and conflict	Many stakeholders, years of development, changing business	Traceability, prioritisation, empowered change control board

Estimation failure	Novel elements, hidden integration work, external dependencies	Multiple estimation methods, ranges not single dates, phased commitments
Late integration surprises	Subsystems built in isolation on differing assumptions	Interface control, continuous integration, incremental delivery
Quality and defect control	Defects and test combinations grow beyond manual control	Defect tracking, root-cause analysis, automated regression testing
Key-person and turnover risk	Knowledge concentrates in a few long-serving individuals	Documentation, pairing, succession and cross-training
Scope creep and gold-plating	Many requesters, weak central authority	Baseline scope, priced change requests, visible backlog

Two further problems deserve separate emphasis because they are so often decisive. One is human: motivation, turnover and the loss of key people. In a large, long project, irreplaceable knowledge quietly concentrates in a few individuals; if they leave, burn out or are reassigned, the project loses not workers but memory. Countermeasures — documentation that is actually maintained, pair work, cross-training and sane workloads — are management responsibilities, not personal virtues. The other is governance failure: optimistic reporting travelling upward, so

that each level of management hears a slightly better story than the level below, and senior decision-makers learn of a crisis only when nothing can be done. Large organisations counter this with objective measures, independent reviews and a culture in which early bad news is treated as valuable. Underlying all of these is a single principle: in large systems, management is not overhead on the engineering — a large part of the engineering simply is management, because the technical work cannot succeed unless the human system producing it is organised, informed and honest.

Conclusion

To conclude, the delayed project of part (a) was not failed by its programmers but by the absence of the functions project management exists to supply: evidence-based estimation, a real plan with dependencies and buffers, tracking against objective milestones, early risk and change control, and honest communication about the scope–time–cost balance. In large-scale systems, part (b), these functions become harder and more essential at once, as communication overhead, requirements volatility, integration risk, quality control and key-person dependence all grow with size. The consistent lesson of software engineering research and practice is that delay is rarely an accident; it is the predictable output of unmanaged work, and it is prevented — when it is prevented — by management practised as an engineering discipline.

USMAN NOTES www.usmannotes.com

Question 4: System Engineering in a Hospital System; Elements of an Online Shopping System (20 Marks)

Question: (a) A hospital system integrates multiple subsystems. Explain how system engineering principles apply. (b) Analyze the elements of a system in an online shopping platform.

Introduction

Software engineering builds the software; system engineering concerns the whole — hardware, software, people, procedures, data and facilities working together to achieve a purpose in a real environment. The distinction matters most exactly where this question places it: in a hospital, where a failure is measured not in inconvenience but in patient safety, and in a large online shopping platform, where millions of interacting elements must behave as one dependable system. Part (a) explains how system engineering principles apply to a hospital system that integrates many subsystems. Part (b) analyses the elements of a system using an online shopping platform as the example, identifying each element and showing how the whole exhibits properties that no element possesses alone.

Part (a): The Hospital as a System of Systems

A modern hospital system is not one program but an integration of many subsystems: patient registration and admission–discharge–transfer, the electronic health record, laboratory information, radiology and imaging, pharmacy and medicine dispensing, operating-theatre and bed management, billing and insurance, appointment scheduling, and interfaces to medical

devices such as monitors and analysers. System engineering begins, before any of these is built, with the definition of the whole system's purpose and its stakeholders: patients, doctors, nurses, pharmacists, laboratory staff, administrators, insurers and regulators, whose needs sometimes conflict and must be balanced explicitly. Requirements are then defined at the system level — for example, that a patient's allergy information recorded anywhere must be visible everywhere a medicine is prescribed — and allocated down to subsystems, so that each subsystem knows not only its own functions but the obligations it owes the whole. This allocation and the management of the interfaces between subsystems are the heart of the discipline, because in integrated systems most failures occur at the boundaries: the laboratory result that arrives in a format the record system misreads, the patient identified differently in two subsystems, the pharmacy system that does not learn that a prescription was changed.

System engineering therefore insists on interface definition and standards, on a single authoritative identity for the patient across subsystems, and on interoperability standards of the kind health informatics has developed for exchanging clinical data. It insists equally on system-level qualities that no single subsystem can deliver alone. Reliability and availability must be engineered for the whole service — the emergency department cannot wait because one server is down — which drives decisions about redundancy, backup and graceful degradation. Safety is treated as a system property: hazards are analysed at the level of the whole process, so that, for example, look-alike medicine names, interrupted nurses and default doses are considered together rather than blamed on one screen. Security and confidentiality likewise span subsystems, because patient data flowing across many components must be protected at every hop and access controlled by role throughout. Verification and validation are

also system activities: subsystems are tested individually, then integrated step by step, then the whole is validated in realistic clinical scenarios, including the failures — power loss, network loss, a subsystem offline — that the hospital must survive without harming patients.

Finally, system engineering manages the life of the whole: configuration control across many interdependent components, so that upgrading one subsystem does not silently break three others; change impact analysis before any modification; training and procedures treated as parts of the system, because a technically correct system operated by untrained staff in a chaotic process is not a working hospital system at all; and eventual replacement planned subsystem by subsystem without stopping care. The principle that unifies all of this is emergence: the behaviour of the integrated hospital — whether a deteriorating patient is noticed in time, whether the right medicine reaches the right patient — is a property of the interactions, and it can be engineered only by somebody whose responsibility is the whole. That somebody is the system engineer.

Part (b): Elements of a System in an Online Shopping Platform

A system, in the standard definition used in this course, is a collection of interrelated elements organised to achieve a purpose, operating within an environment and across a boundary that separates what is inside the system from what is outside it. An online shopping platform displays every classical element clearly. Its purpose is to let customers find, order and receive goods while letting sellers reach a market and the platform earn revenue. Its inputs are customer searches, orders and payments, sellers' product data, stock updates and delivery information. Its processes transform those inputs: catalogue search and recommendation,

basket and checkout handling, payment authorisation, order fulfilment, dispatch and returns processing. Its outputs are confirmed orders, payments to sellers, deliveries to customers, invoices, notifications and management information. Its storage elements hold the product catalogue, customer accounts, order histories and stock records. A feedback and control element — sales dashboards, fraud detection, stock alerts, customer ratings and review monitoring — compares actual behaviour with targets and triggers correction, exactly as a control loop does in any engineered system.

Table 1: Elements of a system, analysed in an online shopping platform

System element	In the online shopping platform	Example of interaction
Input	Searches, orders, payments; seller product and stock data	A search triggers catalogue and stock processes
Process	Search, checkout, payment, fulfilment, returns	Checkout calls payment, stock and notification processes
Output	Order confirmations, deliveries, seller payments, invoices	A confirmed order produces dispatch and payment outputs
Storage	Catalogue, accounts, orders and stock databases	Every process reads and updates shared stored data

Control / feedback	Fraud checks, stock alerts, ratings, sales dashboards	Low-stock feedback triggers reordering; fraud scores block orders
Boundary	Platform's own functions versus banks, couriers and sellers' systems	Payment crosses the boundary to the bank's system and returns an authorisation
Environment	Customers, sellers, banks, couriers, regulators, the internet	A courier failure or a new tax rule changes system behaviour from outside
Subsystems	Storefront, catalogue, payment, warehouse, recommendation	Each is a system in itself, with its own elements

USMAN NOTES www.usmannotes.com

Three analytical points complete the picture. First, the platform is a hierarchy of subsystems: the payment subsystem, the catalogue subsystem and the logistics subsystem are each systems in their own right, with their own inputs, processes, outputs and controls, and the platform is their system of systems — so the boundary one draws depends on the question one is asking. Second, the platform shows emergence and interdependence: trust, the property on which the whole business rests, belongs to no single element; it emerges from accurate descriptions, honest reviews, reliable delivery and painless refunds interacting well, and it collapses if any

one of them fails persistently. Likewise a change in one element propagates — a promotion that trebles orders is simultaneously a warehouse problem, a courier problem and a customer-service problem — which is why changes are analysed for system-wide impact. Third, the environment is active, not scenery: competitors' prices, payment regulations, sales-tax rules and seasonal demand all press on the system from outside its boundary, and the feedback elements exist precisely so the system can sense and adapt. To analyse a system, in short, is to identify its purpose, elements and boundary, trace the interactions, and judge the behaviour of the whole rather than the performance of any part in isolation.

Conclusion

To conclude, the hospital of part (a) shows system engineering in its most demanding form: stakeholder needs defined for the whole, requirements allocated to subsystems, interfaces and interoperability controlled, and safety, reliability, security and validation treated as properties of the integrated system rather than of any component. The shopping platform of part (b) shows the analytical vocabulary — purpose, inputs, processes, outputs, storage, control and feedback, boundary, environment and subsystems — at work on a familiar example, revealing emergence and interdependence as the facts that make systems thinking necessary. Software is built component by component; systems succeed or fail as wholes, and engineering them as wholes is the discipline this question describes.

Question 5: Analysis Principles Applied to a University Management System (20 Marks)

Question: Apply analysis principles to a university management system.

Introduction

Analysis is the stage of software engineering at which the developer stops asking "how shall we build it?" and concentrates entirely on "what must it do, for whom, and how will we know?" The output of analysis is a model of the problem — of the information the system must hold, the functions it must perform and the behaviour it must show — that the client can validate and the designer can build from. This question asks us to apply the recognised analysis principles to a concrete and familiar problem: a university management system of the kind an open university such as AIOU needs, covering admission, enrolment, course and tutor allocation, assignment submission and marking, examinations and results, and student support for very large numbers of distance learners. The application below follows the classical principles set out in the course text: understanding the problem domain, modelling information, function and behaviour, partitioning the problem, and examining the essence before the implementation.

The Analysis Principles Themselves

The principles can be stated briefly before they are applied. First, the information domain of the problem must be understood: what data enters the system, what data it must store and maintain, what data flows between its parts, and what information it must produce. Second,

the functions that transform that information must be defined, at a level of detail that lets each function be checked against a real need. Third, the behaviour of the system — how it responds to events over time, the states it passes through — must be modelled, because many requirements are about sequence and timing rather than about data or function alone. Fourth, the models must be created in layers, partitioning the problem: a high-level view that fits on one page, decomposed step by step into detail, so that complexity is managed rather than merely documented. Fifth, analysis must move from essence to implementation: the analyst models what the system must do in a way that is true regardless of technology, and only later are implementation choices — a particular database, language or platform — allowed to shape the design. Underlying all five is the discipline of representation and review: the model is written in standard notations, examined with users, and corrected until both sides agree that it describes the problem faithfully.

Understanding the Problem: The University Domain

Applied to our system, analysis begins with the problem, not the computer. The analyst studies how the university actually works: a student applies for admission to a programme, eligibility is checked, fees are verified, enrolment in courses follows, study material is dispatched or made available online, tutors are appointed for groups of students, assignments are submitted and marked, examinations are scheduled at centres, results are compiled and degrees are awarded. Interviews with admission staff, tutors, examination staff and students, together with the university's existing forms, prospectus rules and regulations, reveal the real rules — credit-hour limits per semester, prerequisites, re-sit rules, deadlines after which late assignments cannot be accepted — that the software must embody. This domain study also exposes the problems the

system exists to solve: lost paper records, slow result compilation, students unable to learn their status, and regional offices working from inconsistent copies of the same data. A system analysed without this study would automate the forms while missing the rules, which is the most common and most expensive analysis failure.

Information Modelling: What the System Must Remember

The information domain is modelled first with an entity-relationship view of the data the university must hold. The principal entities are Student, Programme, Course, Enrolment, Tutor, Assignment, Submission, Examination and Result, together with Fee Record and Study Centre. The relationships carry much of the meaning: a student enrols in many courses and a course is taken by many students, resolved through the Enrolment entity, which itself carries the semester and status; a course has many assignments; a submission belongs to exactly one assignment and one student; a tutor is allocated to many enrolments in a region; a result is recorded for one student in one course in one examination session. Attributes, keys and integrity rules follow from the domain study: a student identifier must be unique and permanent, a result must never exist for a student not enrolled in the course, and marks must lie within the range the assignment or paper allows. Producing this model with the university's staff — walking through real cases such as a student who changes programme mid-study — validates the information domain before a single table is designed.

Functional and Behavioural Modelling

Function is modelled with data flow diagrams, partitioned from a context view downward. At the highest level the whole system is one process exchanging data with students, tutors, banks

and regional offices; the level below separates the major functions — Manage Admissions, Manage Enrolment, Allocate Tutors, Process Assignments, Conduct Examinations and Compile Results — each then decomposed until every bubble is a single, checkable task, such as verifying an assignment submission against its deadline. Behaviour is modelled with state diagrams for the entities whose lives have meaningful stages. An enrolment, for instance, passes from Applied to Admitted, to Enrolled, to Under Study, to Assessed, to Completed or Failed, with defined events — fee confirmed, deadline passed, result declared — driving each transition, and forbidden transitions, such as a result being declared before assessment, ruled out by the model itself. An assignment submission similarly moves from Issued to Submitted to Marked to Returned, with a Late state governed by explicit rules. These behavioural models frequently reveal requirements that interviews miss: what happens to a submission whose tutor is replaced, or to an enrolment when a course is withdrawn — questions the model forces the university to answer in analysis, while the answers are cheap.

Table 1: Analysis principles applied to the university management system

Analysis principle	How it is applied here	Product of the work
Understand the information domain	Entities, relationships, keys and data rules identified with university staff	Entity-relationship model and data dictionary
Model the functions	Data flow diagrams partitioned from context level to detailed processes	Levelled DFD set with process specifications

Model the behaviour	State diagrams for enrolment, submission and result life cycles	State-transition diagrams and event lists
Partition the problem	One-page context view decomposed level by level	Manageable, reviewable models at each level
Essence before implementation	Models state what must happen, independent of platform or language	Technology-free analysis model for validation
Represent and review	Standard notations walked through with users using real cases	Agreed, corrected model forming the basis of the SRS

USMAN NOTES www.usmannotes.com

Two practical disciplines complete the application. The first is the data dictionary and process specification: every data flow named in the diagrams is defined, and every lowest-level process is specified precisely enough — conditions, calculations, exceptions — that two readers cannot interpret it differently. The fee-verification process, for example, must state what counts as proof of payment, how partial payments are treated and what the student is told in each case. The second is validation of the whole model against the original problem: traceability from each stated need of the university to the model elements that satisfy it, and structured walkthroughs in which admission, tutor and examination staff follow their own real scenarios through the diagrams. Requirements that prove contradictory — a deadline rule that conflicts with a re-sit

rule, for instance — are resolved here, in the model, rather than in code. Only when the analysis model survives this scrutiny is it frozen as the basis of the software requirement specification and handed to design.

Conclusion

To conclude, applying the analysis principles to a university management system means modelling the problem in three complementary views — the information the university must hold, the functions that transform it, and the behaviour of its key entities over time — partitioning those models from a context view into checkable detail, keeping them free of implementation choices, and validating them with the people who own the problem. The reward is proportionate to the effort: a system whose rules were agreed before they were coded, whose exceptions were discovered while they were still sentences in a document, and whose design can proceed from a model that students, tutors and administrators have all recognised as their university. Analysis, done this way, is not a preliminary to the real work; it is the part of the work on which everything else depends.

USMAN NOTES www.usmannotes.com