

Introduction to Software Engineering

Consolidated Reference Note

COVERAGE

Sections 1–5 of the lecture series, consolidated with Sommerville (10th ed.) Chapters 1–6 and the tutorial series (Tutorials 1, 3–8).

STRUCTURE

Seven chapters · 60+ original diagrams
Comparison tables · Worked UML examples
Glossary and quick-reference appendices

University of Colombo School of Computing
Course lecturer: Prof. K. P. Hewagamage

This is a study reference compiled from lecture slides, prescribed readings and tutorial material. It is not an official course document.

How to use this note

This note consolidates the whole SCS1303 syllabus into one continuous document. Material that appeared in more than one place — and a great deal did — has been merged and placed where it belongs logically rather than where it happened to be taught.

What was consolidated

Topic	Appeared in
Verification and validation	Section 2 (as a process model), Section 4 (requirements validation), Tutorial material. Merged into §2.6 with a cross-reference from Chapter 4.
Prototyping	Section 2 (process model), Section 4 (elicitation technique). Merged into §2.7; the elicitation use is cross-referenced.
Plan-driven vs. Agile	Opens Sections 2, 3, 4 and 5 separately. Established once in §2.3; later chapters carry only a short culture-specific table.
Incremental vs. iterative	Section 2 (twice) and Section 3. Resolved once in §2.5.1.
Cost of late defects	Sections 1, 2 and 4. Stated once in §1.3 with one figure.
Software types and domains	Section 1 slides plus Pressman-style lists in the same deck. Merged into one taxonomy, §1.2.
UML diagrams	Introduced in Section 4, taught in Section 5, drilled in Tutorials 3, 4, 7, 8. All notation collected in Chapter 7.
Code review and inspection	Section 2 (as a V&V technique) and Section 6 (walkthrough, inspection, pull requests). The instruments are defined once in §2.6; Chapter 8 covers the implementation-side workflow and cross-references back.
Refactoring and technical debt	Section 3 (an XP practice), Section 5 (a design pitfall) and Section 6 (repaying debt). Consolidated in §8.4.5 with cross-references.
SOLID and design principles	Section 5 states cohesion, coupling and information hiding; Section 6 states SOLID. The mapping between them is given once, in §8.4.4.
CI/CD	Section 2 (DevOps), Section 6 (GitHub Actions, PR checks) and Section 7 (the test pipeline). Defined in §2.12; Chapters 8 and 9 refer to it rather than restating it.
Testing stages	Section 2 (as a V&V activity) and Section 7 (in full). §2.6 keeps the process-level view — four levels and what each catches — and Chapter 9 gives the complete treatment including release testing. Alpha/beta is defined once, in §9.7.
Test-Driven Development	Section 3 (an XP practice) and Section 7 (in full). The Red–Green–Refactor cycle is drawn once in §3.4; §9.5 adds the worked example and the benefits.
Regression testing	Mentioned in Sections 2 and 7. Defined once in §9.5.
AI in software engineering	Closes every lecture section. Kept as one subsection per chapter — they genuinely differ.

Reading paths

- **First pass / learning from scratch.** Read Chapters 1–9 in order. The running example (a university student portal) threads through Chapters 5–9, so the material builds on itself: the design of Chapters 5–7 is turned into code in Chapter 8 and then verified in Chapter 9.
- **Revision.** Read the chapter openers (learning outcomes), then every **Exam pointer** box, then the comparison tables in Appendix B.
- **Tutorial preparation.** Chapter 7 is the notation reference; Appendix A models one system four different ways, which is exactly the exercise Tutorials 3, 4, 7 and 8 set.

Conventions used

Element	Meaning
Blue box	A definition to be reproduced accurately. Standard-body wording is quoted where one exists.
Amber box	The key idea of a topic — the one sentence to retain if nothing else.
Orange box	An exam pointer: a distinction that is commonly examined or commonly confused.
Grey box	A worked example or applied illustration.
✓ / ✗	An advantage or good practice / a disadvantage or anti-pattern.

Source references

- Sommerville, I. *Software Engineering*, 10th ed., Addison-Wesley, 2015 — Chapters 1–6.
- Pressman, R. S. *Software Engineering: A Practitioner's Approach*, 9th ed., McGraw-Hill, 2020.
- Schwaber, K. & Sutherland, J. *The Scrum Guide*, 2020 edition.
- Martin, R. C. *Clean Code*, Prentice Hall, 2008 — readable, maintainable code.
- ACM/IEEE-CS *Software Engineering Code of Ethics and Professional Practice*.
- IEEE Std 610.12 (glossary); ISO/IEC/IEEE 29148:2018 (requirements engineering).
- OMG *Unified Modeling Language* specification, version 2.5.1.

Contents

How to use this note	1
1 Software and Software Engineering	10
1.1 What is software?	10
1.1.1 Generic versus bespoke software	10
1.2 Types and domains of software	11
1.2.1 Classification 1 — system versus application software	11
1.2.2 Classification 2 — application domains	11
1.3 Why software is different	11
1.3.1 The lifetime cost profile	12
1.3.2 How software fails compared with hardware	12
1.4 The software crisis and the birth of the discipline	13
1.4.1 The questions that motivated the discipline	13
1.4.2 Real project failures	13
1.4.3 The communication problem	13
1.5 What is software engineering?	14
1.5.1 How software engineering answered the crisis	14
1.5.2 Related roles distinguished	14
1.5.3 Attributes beyond coding ability	14
1.6 The software development life cycle	15
1.6.1 Who builds software — project team roles	15
1.6.2 Project success and failure rates	15
1.7 Software quality	16
1.8 Professional and ethical responsibility	16
1.8.1 The ACM/IEEE-CS Code of Ethics — eight principles	16
1.9 The modern software landscape	16
1.9.1 How software is delivered today	17
1.9.2 Core principles of software engineering	17
1.9.3 AI and software engineering	17
1.10 Chapter summary — standard question and answer	18
2 Software Processes	19
2.1 What is a software process?	19
2.1.1 Why a defined process matters	19
2.2 The four fundamental process activities	20
2.3 Two cultures: plan-driven and agile	20
2.4 The Waterfall model	21
2.4.1 The five phases in detail	21
2.4.2 Waterfall — strengths, limitations, and when to use it	23
2.5 Incremental development	24
2.5.1 Incremental versus iterative — resolving the confusion	24
2.5.2 Incremental delivery	24
2.5.3 Benefits and problems	25
2.6 Verification, validation and the V-model	25
2.6.1 The V-model	26
2.6.2 Stages of testing	26
2.6.3 Inspections and reviews — V&V without execution	27

2.7	Prototyping	27
2.7.1	Throwaway prototyping	28
2.7.2	Evolutionary prototyping	28
2.8	The Spiral model	29
2.8.1	Risk as the central driver	29
2.9	Rapid Application Development (RAD)	30
2.10	Reuse-oriented software engineering	30
2.11	The Unified Process	31
2.12	DevOps	32
2.13	Coping with change and process improvement	33
2.13.1	Why change is inevitable	33
2.13.2	Software process improvement	33
2.14	AI in software processes	34
2.15	Master comparison of process models	35
3	Agile Software Development	36
3.1	Why agile?	36
3.2	The Agile Manifesto	36
3.2.1	The twelve principles	37
3.2.2	Values, principles and practices	37
3.3	Scrum	38
3.3.1	Five values	38
3.3.2	Three roles	38
3.3.3	Three artifacts	39
3.3.4	Five events	39
3.3.5	The sprint cycle	39
3.3.6	The Definition of Done	40
3.3.7	Tracking a sprint — the burndown chart	40
3.3.8	Strengths and challenges	40
3.4	Extreme Programming	41
3.4.1	Five values	41
3.4.2	The twelve practices	41
3.4.3	Test-driven development	42
3.4.4	User stories and INVEST	42
3.4.5	Strengths and fit	42
3.5	Kanban	43
3.5.1	The Kanban board	43
3.5.2	Work-in-progress limits	43
3.5.3	Six core practices	44
3.5.4	Scrum versus Kanban	44
3.6	Lean software development	44
3.6.1	Seven principles	45
3.6.2	Seven wastes in software	45
3.7	Other agile methods	45
3.8	Choosing an agile approach	46
3.8.1	Agile at scale	46
3.9	Agile tools	46
3.9.1	Jira	47
3.9.2	Trello	47
3.9.3	Other tools worth knowing	48
3.10	AI in agile development	48

4	Requirements Engineering	50
4.1	Why requirements matter	50
4.2	Requirements engineering in both cultures	51
4.3	What is a requirement?	51
4.3.1	The lifecycle of a requirement	51
4.4	The three dimensions of requirements	52
4.4.1	Functional requirements	52
4.4.2	Non-functional requirements	52
4.4.3	Domain requirements	53
4.5	Requirement quality	54
4.5.1	The danger of ambiguity	54
4.5.2	Sharpening a requirement	54
4.5.3	Properties of a good requirement	54
4.6	The requirements engineering process	54
4.6.1	Feasibility study	54
4.7	Requirements elicitation	55
4.7.1	Why elicitation is difficult	55
4.7.2	Stakeholders	56
4.7.3	Fact-finding and domain analysis	56
4.7.4	Defining the problem and the scope	57
4.7.5	Four elicitation activities	57
4.7.6	Elicitation techniques	57
4.7.7	The discovery toolkit, by category	58
4.7.8	Common challenges	58
4.8	Requirements analysis and modelling	58
4.8.1	Requirements classification	58
4.9	Prioritising requirements	59
4.9.1	MoSCoW	59
4.9.2	Other prioritisation techniques	59
4.10	Requirements specification	60
4.10.1	Writing one requirement well	60
4.10.2	The Software Requirements Specification	60
4.10.3	The product backlog — agile specification	61
4.10.4	SRS versus product backlog	61
4.10.5	Common problems in requirements documents	62
4.11	Requirements validation	62
4.11.1	The 5C check	62
4.11.2	Validation techniques	62
4.12	Requirements management	63
4.12.1	Why requirements change	63
4.12.2	Two pillars	63
4.12.3	The change management process	63
4.12.4	Traceability	63
4.13	AI in requirements engineering	64
4.13.1	Cautions	64
4.14	Common pitfalls in requirements engineering	64
5	Software Design	65
5.1	What is software design?	65
5.1.1	What makes a design good?	66
5.2	The design journey — top down	66
5.2.1	Level 1 — architectural design	66

5.2.2	Level 2 — component design	66
5.2.3	Level 3 — interface design	67
5.2.4	Level 4 — data design	67
5.3	Core design principles	68
5.3.1	Abstraction	68
5.3.2	Modularity	69
5.3.3	Cohesion	69
5.3.4	Coupling	69
5.3.5	Information hiding	70
5.3.6	Separation of concerns — the shared goal	70
5.4	Major design approaches	70
5.4.1	1. Procedural / workflow design	70
5.4.2	2. Object-oriented design	71
5.4.3	3. Component-based design	71
5.4.4	4. Layered / tiered design	71
5.4.5	5. Service-oriented / API-based design	72
5.4.6	6. Event-driven design	72
5.4.7	7. Functional and data-centred design	72
5.4.8	Choosing an approach	73
5.5	Design in both cultures	73
5.5.1	Plan-driven design — complete blueprint before build	73
5.5.2	Agile design — enough design, then continuous improvement	73
5.5.3	Side by side	74
5.6	Design specification	74
5.6.1	Design specification in agile	75
5.7	Tools and AI in software design	75
5.8	Common design pitfalls	76
6	Software Architecture	77
6.1	What is software architecture?	77
6.1.1	Why architecture must come first	77
6.1.2	Architectural design decisions	77
6.1.3	Architecture and non-functional requirements	78
6.1.4	Architectural views	78
6.2	Architectural patterns	78
6.2.1	Pattern 1 — Layered architecture	78
6.2.2	Pattern 2 — Client-server	79
6.2.3	Pattern 3 — Model-View-Controller	80
6.2.4	Pattern 4 — Repository	80
6.2.5	Pattern 5 — Pipe and filter	81
6.3	Microservices	81
6.4	Application architectures	82
6.4.1	Transaction-processing systems	82
6.4.2	Language-processing systems	82
7	System Modelling and UML	84
7.1	Why we model	84
7.1.1	Four perspectives	84
7.1.2	UML	85
7.2	Flowcharts	85
7.3	Use case diagrams	86
7.3.1	Notation	86

7.3.2	The three relationships between use cases	86
7.3.3	Worked example — Student Portal	86
7.3.4	How to build a use case diagram from a scenario	87
7.4	Class diagrams	88
7.4.1	The class box	88
7.4.2	Relationships	88
7.4.3	Multiplicity	88
7.4.4	Association classes	89
7.4.5	Worked example — Student Portal	89
7.4.6	Building a class diagram from a scenario	90
7.5	Sequence diagrams	91
7.5.1	Lifelines and activation	91
7.5.2	Creation and destruction	91
7.5.3	The four message types	92
7.5.4	Three rules that govern the drawing	92
7.5.5	Worked example 1 — a login, start to finish	92
7.5.6	Combined fragments	92
7.5.7	Worked example 2 — an ATM withdrawal with its branch	93
7.5.8	Red-pen corrections	94
7.5.9	Drafting diagrams as text	94
7.6	Activity diagrams	94
7.6.1	Notation	95
7.6.2	Worked example — applying for a library card	95
7.6.3	Worked example — fork and join	96
7.6.4	Swimlanes	96
7.7	State machine diagrams	97
7.8	Component and deployment diagrams	97
7.9	Context models and data flow diagrams	97
7.10	Choosing the right diagram	98
8	Software Implementation	99
8.1	What is implementation?	99
8.1.1	Implementation in the lifecycle	99
8.1.2	A misconception about implementation	99
8.2	Programming languages	100
8.2.1	How languages evolved	100
8.2.2	Programming paradigms	100
8.2.3	The modern landscape	101
8.2.4	Choosing a language	101
8.3	The development environment	102
8.3.1	Integrated development environments	102
8.3.2	Package managers	103
8.3.3	Linters and formatters	103
8.4	Good coding practices	103
8.4.1	Reliability — fault avoidance and fault tolerance	104
8.4.2	Readability — naming	104
8.4.3	Comments and self-documenting code	105
8.4.4	The SOLID principles	105
8.4.5	Technical debt	106
8.5	Code quality and review	106
8.5.1	Walkthrough and inspection	106
8.5.2	Pull requests — the modern workflow	107

8.6	Version control with Git	107
8.6.1	Git concepts	108
8.6.2	The three stages of a file	108
8.6.3	Core commands	109
8.6.4	Branching	109
8.6.5	Commit messages	109
8.6.6	Git versus GitHub	110
8.6.7	A complete workflow	110
8.6.8	Merge conflicts	111
8.7	Open source and licensing	111
8.7.1	The common licences	112
8.7.2	Choosing a licence for your own project	112
8.8	AI in implementation	112
8.8.1	Skills that become <i>more</i> important	113
8.8.2	The tool landscape	113
8.8.3	Prompt engineering	113
8.8.4	What AI does well — and what it gets wrong	114
8.8.5	What the engineer still owns	114
8.9	Chapter summary	115
9	Software Testing	116
9.1	What is software testing?	116
9.1.1	The fundamental limit	116
9.1.2	Testing within verification and validation	117
9.1.3	The testing stages	117
9.2	Black box and white box testing	118
9.3	Test design techniques	118
9.3.1	Equivalence partitioning	119
9.3.2	Boundary value analysis	119
9.3.3	Testing guidelines	119
9.3.4	Use-case based testing	120
9.3.5	The structure of a test case	120
9.4	Development testing	120
9.4.1	Unit testing	121
9.4.2	Test coverage	122
9.4.3	Component and integration testing	122
9.4.4	System testing	123
9.5	Test-Driven Development	123
9.5.1	Regression testing	124
9.6	Release testing	125
9.6.1	Requirements-based testing	125
9.6.2	Performance and load testing	125
9.7	User testing	126
9.7.1	Acceptance testing — the formal gate	126
9.8	Modern testing practice	127
9.8.1	Continuous testing in CI/CD	127
9.8.2	API testing	127
9.8.3	Mutation testing — testing the tests	128
9.9	AI in software testing	128
9.10	Chapter summary	130

A	Worked example: one system, four models	131
A.1	The scenario	131
A.2	Step 1 — extract the model elements	131
A.3	Step 2 — the use case diagram (external view)	132
A.4	Step 3 — the class diagram (structural view)	132
A.5	Step 4 — the sequence diagram (interaction view)	133
A.6	Step 5 — the activity diagram (behavioural view)	134
A.7	Step 6 — comparing the four models	134
B	Quick reference	136
B.1	The one-page map of the subject	136
B.2	Definitions worth reproducing exactly	137
B.3	Distinctions that are commonly confused	138
B.4	Numbers worth remembering	139
B.5	Answer templates for common question types	140
C	Tutorial coverage map	141
C.1	Scenario checklists	141
C.1.1	Any use case scenario	142
C.1.2	Any class diagram scenario	142
C.1.3	Any activity diagram scenario	142
C.1.4	Any test design exercise	142
C.1.5	Any Git or collaboration exercise	143
C.1.6	Any sequence diagram scenario	143
D	Glossary	144

Software and Software Engineering

Learning outcomes for this chapter

- LO1.** Define software and software engineering using recognised standard definitions.
- LO2.** Classify types of software and their application domains with real examples.
- LO3.** Explain the characteristics that make software development uniquely difficult.
- LO4.** Describe the software crisis and how it produced software engineering as a discipline.
- LO5.** Outline the SDLC stages and the roles within a software project team.
- LO6.** Identify the professional and ethical responsibilities of a software engineer.
- LO7.** Describe how modern software is delivered — web, cloud, mobile and AI-assisted.

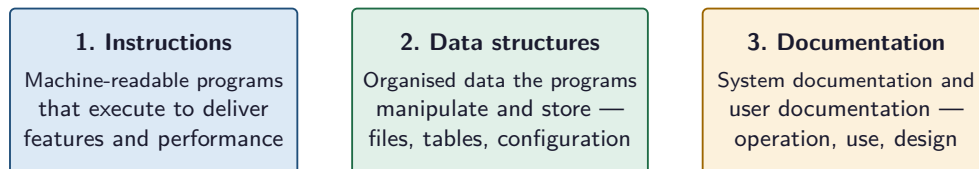
1.1 What is software?

Software — IEEE definition

“Computer programs, procedures, and associated documentation and data pertaining to the operation of a computer system.”

The point of the definition is that *software is not only code*. A program you cannot install, configure, understand or maintain is not a usable product.

Software has three essential components. Delivering only the first is the single most common misunderstanding among beginning developers.



All three are delivered to the customer. Omitting the third is what makes a system unmaintainable long before it becomes unreliable.

Figure 1.1: The three components of software.

1.1.1 Generic versus bespoke software

	Generic (product)	Bespoke (custom)
Built for	A broad market of many customers	One named customer
Who owns the specification	The developer	The customer
How the spec is set	Market research, competitive analysis	Direct negotiation with the client
Cost per user	Low — development cost is amortised	High — one customer bears everything
Resale	Sold many times	Not resaleable
Change control	Developer decides the roadmap	Customer requests changes contractually
Examples	Microsoft Windows, WhatsApp, Photo-shop, Chrome	Hospital management system, payroll system, airline reservations

Exam pointer

The distinction matters because it drives *process choice*. Bespoke systems with a contractual specification lean plan-driven; generic products with an unknown market lean agile and incremental. Every process-model question in Chapter 2 implicitly assumes one or the other.

1.2 Types and domains of software

Two orthogonal classifications are used, and they are often confused. The first asks *what the software controls*; the second asks *what problem domain it serves*.

1.2.1 Classification 1 — system versus application software

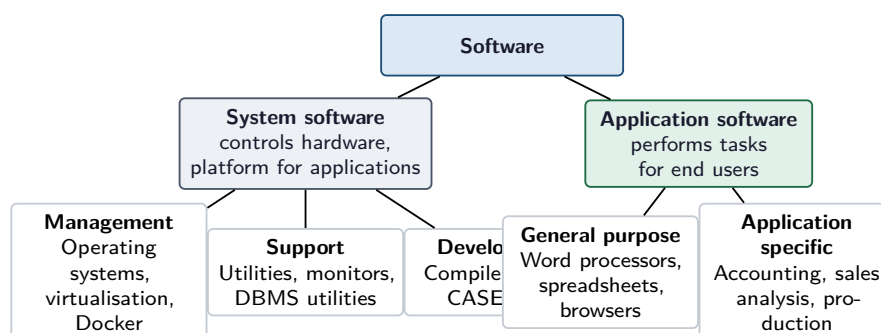


Figure 1.2: System software provides the platform; application software runs on top of it.

1.2.2 Classification 2 — application domains

Domain	Character	Examples
Embedded	Resides in read-only memory; controls a device; hard real-time constraints	Pacemakers, engine control units, ATMs
Web and mobile	Distributed, browser- or handset-delivered, frequent release	Banking apps, e-commerce, social media
Business	Management information systems over large databases	ERP, CRM, payroll, inventory
Scientific / engineering	Numerically intensive modelling, simulation and analysis	Climate models, finite-element analysis
Artificial intelligence	Non-numerical algorithms for problems not amenable to straightforward analysis	Fraud detection, recommenders, LLM assistants
System software	Services other programs rather than end users	Operating systems, compilers, DBMS
Product-line	A family of related products sharing a common core	Banking product suites
Legacy	Old but business-critical; expensive to replace	COBOL mainframe systems

Key idea

Every domain uses the same engineering principles, but weights them differently. A game is best developed through a series of prototypes; a safety-critical control system requires a complete, analysable specification. This is why no single method can be declared “the best”.

1.3 Why software is different

Software is *engineered*, not manufactured, and this single fact explains most of what follows.

Characteristic	Consequence
Difficult to specify completely	Customers do not know what they want until they can see and touch a working system
Requirements change constantly	Like building a house whose design keeps shifting mid-construction
Software is intangible	You cannot see a bug the way you can see a crack in a bridge; progress is hard to observe
Exhaustive testing is impossible	The combinations of inputs, states and paths are effectively unbounded
Small changes have large effects	A few lines can change system-wide behaviour
Replication is trivial	There is no manufacturing constraint on growth, so systems accrete complexity
The discipline is young	Measurable, well-calibrated techniques are still scarce compared with civil or mechanical engineering

1.3.1 The lifetime cost profile

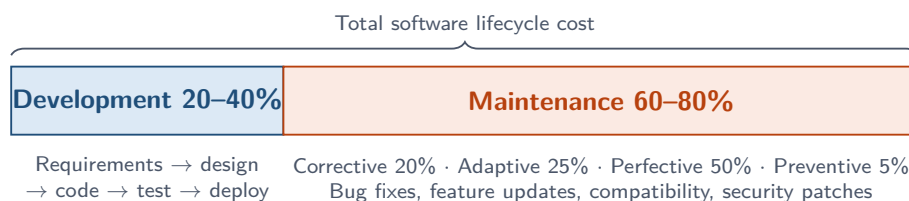


Figure 1.3: Where the money goes. Sources: IEEE Std 1219, Standish Group. For custom software with a long service life, evolution cost commonly exceeds original development cost by a factor of 2–5.

Key idea

Writing code is only the beginning. Engineering software well determines what it costs for the next ten to twenty years. This is why maintainability is treated as a first-class quality attribute and not an afterthought.

1.3.2 How software fails compared with hardware

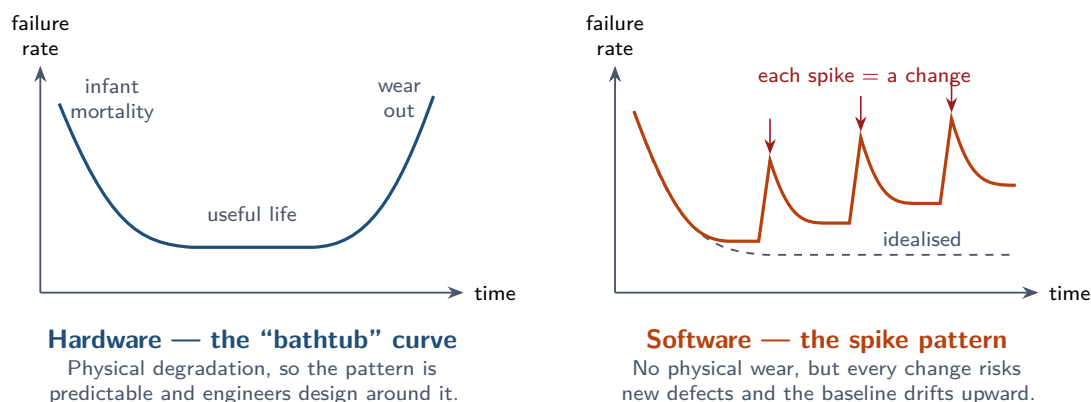


Figure 1.4: Software does not wear out — it *degrades logically*. Complexity accumulates with each modification, so the residual failure rate never returns to its previous level.

Exam pointer

“Software does not wear out” is a standard exam sentence, but the follow-up matters more: *software still deteriorates*, because maintenance introduces defects faster than they are removed. Reviews, testing and clean design are what control that deterioration.

1.4 The software crisis and the birth of the discipline

Through the 1960s hardware became rapidly cheaper and faster while software development did not keep pace. Developers had no structured methods for managing the resulting complexity. The term **software crisis** was formally raised at the **1968 NATO Software Engineering Conference** in Garmisch, Germany.

Symptom	Scale of the problem
Over budget	Around 55% of projects cost more than originally estimated
Late delivery	The average project overshot schedule by 50% or more
Wrong product	Delivered software frequently failed to meet stated user requirements
Unmaintainable	Code was difficult to understand, modify or scale
Never delivered	Roughly a 50% cancellation rate for large software projects
No discipline	No formal engineering methods — ad hoc coding was the norm

1.4.1 The questions that motivated the discipline

These five questions were posed in the 1960s and are still asked today. If you can explain *why* each is hard, you understand the subject.

1. Why does it take so long to finish software?
2. Why are development costs so high?
3. Why can we not find all errors before delivery?
4. Why do we spend so much effort maintaining existing programs?
5. Why is it so hard to measure progress while software is being developed?

1.4.2 Real project failures

Project	What happened	Root cause
UK Home Office National Probation Service system	Ran 70% over its original budget, ending at £118m	Poorly specified requirements plus a restrictive contract
UK Air Traffic Control, Swanwick Centre	Six years late and £180m over budget; near-obsolete before it went operational	Scope creep and technology churn over a long schedule
Boeing 737 MAX (MCAS)	Control-system design flaws contributed to two fatal crashes in 2018–19; global fleet grounded	Safety-critical software developed without commensurate rigour

1.4.3 The communication problem

The most-reproduced illustration in the subject shows one requirement passing through a project and emerging as six different systems. It is a picture of what requirements engineering exists to prevent.

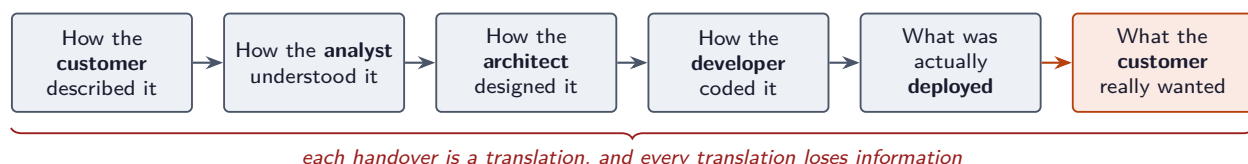


Figure 1.5: The classic communication problem (after Sommerville, Ch. 1).

1.5 What is software engineering?

Software engineering — IEEE Std 610.12

“The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.”

Systematic	Planned and structured, not guesswork or improvisation.
Disciplined	Follows defined principles, standards and good practice.
Quantifiable	Outcomes can be measured, not merely felt or assumed.

Software engineering is concerned with *all* aspects of software production, not only the technical process of development: it also covers project management, tool support, and the methods and theories underlying practice.

1.5.1 How software engineering answered the crisis

Contribution	What it addresses
Requirements engineering	Understanding user needs precisely, using formal and semi-formal methods
Prototyping	Demonstrating early versions to reduce risk and obtain real feedback
Defined engineering process	Structured models (Waterfall, Agile, DevOps) for planning and control
Quality focus	Emphasis on readable, defect-free code that meets expectations
Tool support (CASE)	Automated support for design, testing, documentation and maintenance

1.5.2 Related roles distinguished

	Computer scientist	Software developer	Software engineer
Primary focus	Theory and fundamentals	Coding and building features	Systems and engineering
Scope	The discipline	Project or task level	System-wide, architectural
Thinking style	Analytical, proof-oriented	Tactical — implement what is given	Strategic — plan, model, optimise
Typical activity	Algorithms, data structures, computing theory	Writes code, implements a specification	Applies engineering principles, owns quality attributes

Exam pointer

Two further distinctions are examined regularly. **Software engineering vs. computer science**: computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software. **Software engineering vs. system engineering**: system engineering covers all aspects of computer-based systems development including hardware, software and process engineering — software engineering is a part of that larger process.

1.5.3 Attributes beyond coding ability

1. **Problem solving** — finding the right solution for the right problem, not merely writing code.
2. **Attention to detail** — one misplaced character can crash a system.
3. **Clear communication** — technical ideas must be explained to non-technical stakeholders.
4. **Continual learning** — tools and languages known today may be obsolete tomorrow.
5. **Teamwork** — modern software is built collaboratively, through reviews and shared ownership.

6. **Empathy with end users** — user-centred thinking produces systems people will actually use.

1.6 The software development life cycle

The SDLC is the structural skeleton on which every process model in Chapter 2 is built. The models differ in how they *order*, *repeat* and *overlap* these stages — not in which stages exist.

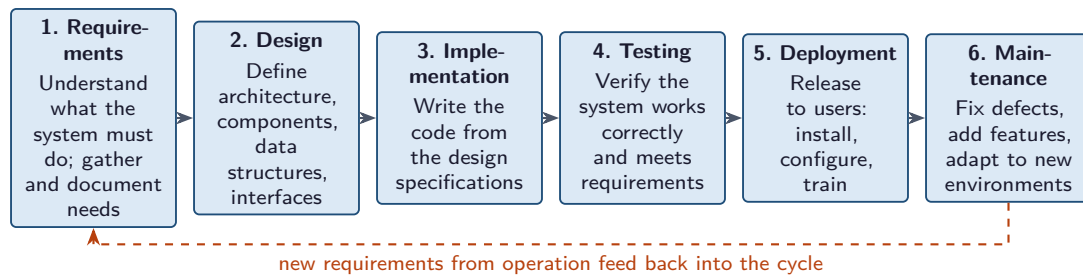


Figure 1.6: The six canonical SDLC stages.

1.6.1 Who builds software — project team roles

Role	Responsibility
Project manager	Planning, timelines, budget and team coordination
Systems analyst	Gathers requirements and defines system specifications
Software architect / designer	Creates system architecture and interface design
Programmer / developer	Writes, builds and integrates the code
QA engineer / tester	Tests the software for quality, correctness and functionality
Technical writer / support	Maintains documentation and assists with administration and user support

1.6.2 Project success and failure rates

Status	Approx. share	Definition
Successful	35–45%	On time, on budget, with full scope
Challenged	40–50%	Delivered, but over budget, late, or missing features
Failed	10–20%	Cancelled, or delivered and never used

Signs of success	Common failure causes
Delivered within agreed time	Unclear or constantly changing requirements
Costs remain within budget	Poor communication between stakeholders
Meets user expectations fully	Unrealistic timelines or budgets set at the start
Motivated, collaborative team	Lack of testing discipline — defects found late
Well-documented, maintainable code	Scope creep — features keep expanding
Realistic, incremental planning	No formal engineering process followed

Success rates improved from the 1990s largely because of better tooling (Jira, Git, CI/CD), agile and DevOps adoption, improved communication tools, and cloud platforms that reduced infrastructure risk. Scope creep, unrealistic expectations and unclear requirements nevertheless persist.

1.7 Software quality

Product quality <i>characteristics of the delivered software</i>	Process quality <i>characteristics of how it was built</i>
Usability — easy and intuitive for its users	Standards followed consistently
Reliability — fails rarely and recovers well	Code reviews and inspections carried out
Efficiency — performs well under normal load	Testing at every stage of development
Maintainability — easy to understand and change	Documentation kept up to date
Security — protects user data and the system	Version control used throughout

Key idea

Product quality is how good the cake is; process quality is how clean and organised the kitchen was. Consistent results require both — good outcomes that depend on which individual happened to write the code are not engineering.

1.8 Professional and ethical responsibility

Software engineering involves wider responsibilities than the application of technical skills. Engineers must behave in an honest and ethically responsible way to be respected as professionals; ethical behaviour is more than simply upholding the law.

Area	Obligation
Confidentiality	Respect the confidentiality of employers and clients, whether or not a formal confidentiality agreement has been signed
Competence	Do not accept work outside your competence; do not misrepresent your level of competence
Intellectual property	Respect copyright, licences and patents — including open-source licence terms
Computer misuse	Do not use technical skills to misuse other people's computers, install malware, or abuse employer resources
Public safety	Safety-critical software demands the highest rigour; lives may depend on it
Honesty	Be transparent about progress, problems and limitations with your team and clients

1.8.1 The ACM/IEEE-CS Code of Ethics — eight principles

Principle	Statement
1. Public	Act consistently with the public interest — welfare and safety come first
2. Client and employer	Serve clients and employers in a manner consistent with the public interest
3. Product	Ensure products and modifications meet the highest professional standards possible
4. Judgement	Maintain integrity and independence in professional judgement
5. Management	Promote an ethical approach to the management of development and maintenance
6. Profession	Advance the integrity and reputation of the profession
7. Colleagues	Be fair to and supportive of colleagues
8. Self	Participate in lifelong learning and promote an ethical approach to the practice

1.9 The modern software landscape

1.9.1 How software is delivered today

Delivery model	Character	Examples
Web-based	Runs in a browser; no installation; accessible from any device	Google Docs, Gmail, online banking
Cloud computing	Software runs on provider-managed remote servers; pay for what you use	Zoom, Netflix, Microsoft 365
Mobile applications	Small screens, variable connectivity, battery constraints	WhatsApp, banking apps, Google Maps
Software as a Service	Subscription rather than purchase; accessed on demand	Spotify, Adobe Creative Cloud

Web-based systems are complex distributed systems, but they follow the same engineering principles as any other software. What differs is the emphasis:

1. **Software reuse** — components, libraries, APIs and services are reused rather than rebuilt.
2. **Incremental and agile development** — requirements change frequently, so delivery is iterative.
3. **Service-oriented architecture** — functionality is exposed as independent services (login, payment gateway) consumed over the network.
4. **Rich interfaces** — AJAX, React and HTML5 support dynamic, interactive interfaces inside the browser.

1.9.2 Core principles of software engineering

Regardless of tools, language or methodology:

Principle	Meaning
Use a defined process	Develop with a structured, managed process. Different systems suit different approaches — no one-size-fits-all method exists.
Ensure dependability	Every system must be reliable, secure and efficient under realistic load.
Start with clear requirements	Know exactly what to build before writing code; requirements engineering prevents wasted effort.
Reuse when possible	Use existing libraries, components and services rather than rebuilding what already works.

1.9.3 AI and software engineering

What AI tools do well	Where engineering remains essential
Generate boilerplate code quickly	Understanding user requirements correctly
Suggest completions for common patterns	Designing reliable, scalable architecture
Explain existing code	Making software maintainable and secure
Speed up routine implementation	Testing thoroughly and systematically
Draft documentation and test cases	Taking professional responsibility for the product

Key idea

AI generates code. Software engineering ensures that code is correct, safe, reliable and maintainable. Professional accountability does not transfer to the tool — the engineer who accepted the suggestion owns the outcome.

1.10 Chapter summary — standard question and answer

Question	Answer
What is software?	Computer programs and associated documentation. Products may be developed for a particular customer (bespoke) or for a general market (generic).
What are the attributes of good software?	It should deliver the required functionality and performance to the user, and should be maintainable, dependable and usable.
What is software engineering?	An engineering discipline concerned with all aspects of software production.
What are the fundamental activities?	Software specification, software development, software validation and software evolution.
Software engineering vs. computer science?	Computer science focuses on theory and fundamentals; software engineering on the practicalities of developing and delivering useful software.
Software engineering vs. system engineering?	System engineering covers all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of that.
What are the key challenges?	Coping with increasing diversity, demands for reduced delivery times, and developing trustworthy software.
What are the costs?	Roughly 60% development, 40% testing of the development budget; for custom software, evolution costs usually exceed development costs.
What are the best techniques?	There is no universal answer. Games are best built through prototypes; safety-critical control systems need a complete, analysable specification.
What difference has the web made?	Availability of software services and highly distributed service-based systems, with major advances in software reuse.

Software Processes

Learning outcomes for this chapter

- LO1.** Explain what a software process is and why a defined process matters.
- LO2.** Distinguish between plan-driven and agile development cultures.
- LO3.** Describe the Waterfall model — phases, strengths and limitations.
- LO4.** Explain the Incremental, Iterative, V-model, Prototyping, Spiral, RAD and Reuse-oriented models.
- LO5.** Explain the Unified Process as a hybrid bridging both cultures.
- LO6.** Identify the four core process activities present in every software process.
- LO7.** Describe the DevOps culture and its role in modern software delivery.
- LO8.** Recognise how AI is beginning to influence software development processes.

2.1 What is a software process?

Software process

A structured set of activities required to develop a software system. It defines *what* to do, *in what order*, *who* is responsible, and *how* to verify it has been done correctly.

Every process description, however informal, contains four things:

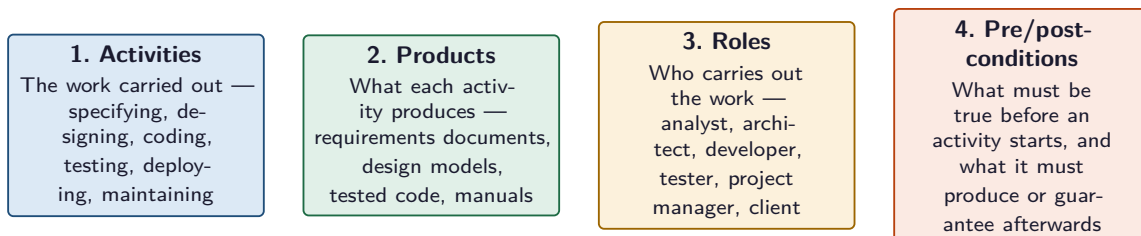


Figure 2.1: The four elements of any software process description.

2.1.1 Why a defined process matters

Benefit	Explanation
Common understanding	Writing the process down creates a shared language. Everyone understands the activities, responsibilities and constraints — no hidden assumptions, no working at cross-purposes.
Consistency and quality	A defined process reduces defects and makes quality repeatable. Good results should not depend on which developer happens to be working today.
Measurable progress	Milestones, deliverables and checkpoints make progress visible. Without them you cannot tell whether a project is on track.
Adaptability	Having a range of models lets a team recognise that different systems need different approaches — and choose deliberately.
Professional practice	A defined process is what separates professional engineering from ad hoc programming.

2.2 The four fundamental process activities

Every process model in this chapter is a different arrangement of the same four activities. Learn these first; the models then become variations rather than separate things to memorise.

Activity	What it establishes	Also called
Software specification	What the system should do and the constraints on its operation	Requirements engineering (Ch. 4)
Software development	Design and implementation of the system that meets the specification	Design (Ch. 5), implementation (Ch. 8)
Software validation	That the software does what the customer requires	Verification and validation (§2.6)
Software evolution	That the software changes to reflect changing customer and market needs	Maintenance

Exam pointer

If asked “what are the fundamental software engineering activities?”, the expected answer is exactly these four: *specification*, *development*, *validation*, *evolution*. Process models differ in how these four are ordered, interleaved and repeated — not in whether they occur.

2.3 Two cultures: plan-driven and agile

Every process model belongs to one culture or deliberately blends both. This distinction is established once here and reused in Chapters 3, 4 and 5.

Plan-driven culture	Agile culture
Plan everything upfront, before coding begins	Embrace change — requirements will evolve
Requirements, design and schedule fixed early	Planning is incremental and continuous
Progress measured strictly against the plan	Working software delivered frequently in short cycles
Change is controlled and expensive	Customer collaboration over contract negotiation
Emphasis on documentation and formal reviews	Self-organising, cross-functional small teams
Best when requirements are stable and well understood	Best when requirements are uncertain or evolving

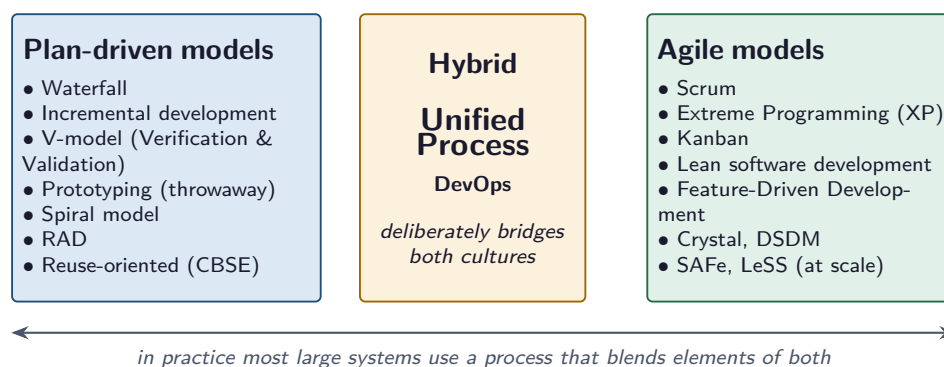


Figure 2.2: Process models mapped to their culture.

2.4 The Waterfall model

Waterfall model

The first formally published software process model (Royce, 1970). Development is divided into distinct phases that flow downward like a waterfall. Each phase must be completed and its output formally approved before the next begins.

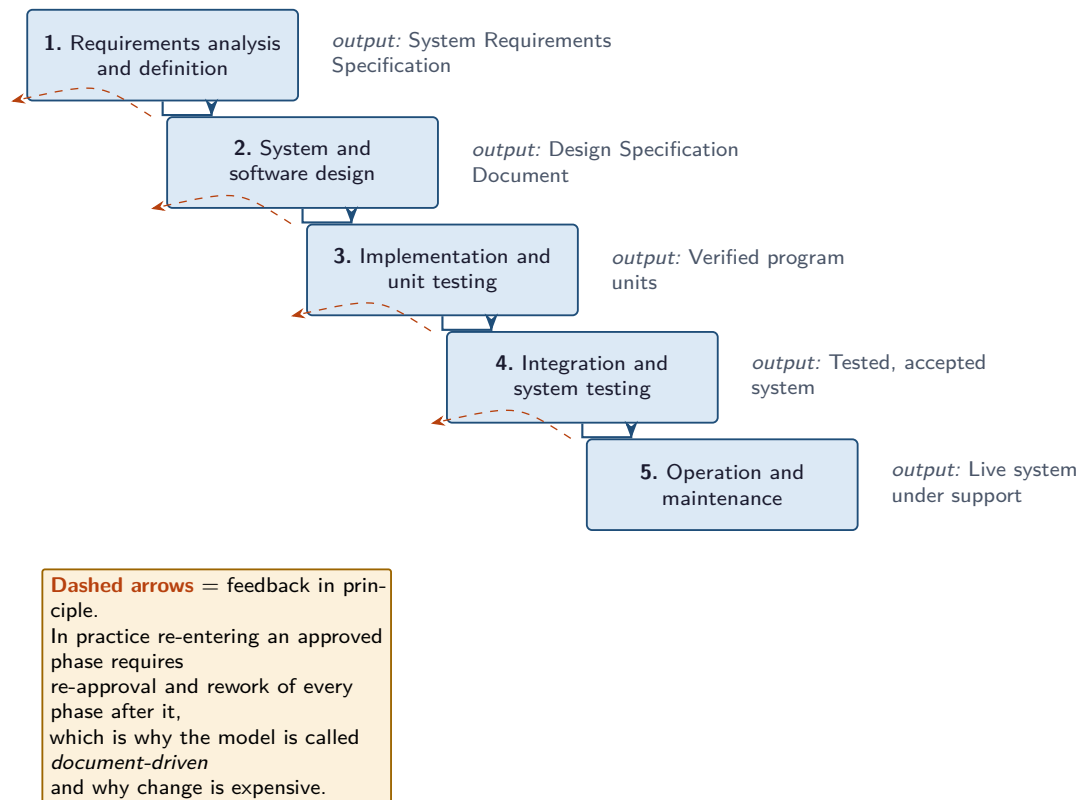


Figure 2.3: The Waterfall model. Royce himself observed in 1970 that pure sequential development is “risky and invites failure” — the model was a starting point, not the final answer.

2.4.1 The five phases in detail

Phase 1 — Requirements analysis and definition

The system’s services, constraints and goals are established by consultation with users and defined in detail as a complete system specification.

Activities	Deliverables
Identify all stakeholders — users, customers, operators, regulators	System Requirements Specification (SRS)
Gather requirements: interviews, workshops, observation, review of existing systems	Functional requirements list
Define functional requirements (what the system must do)	Non-functional requirements list
Define non-functional requirements: performance, security, reliability, usability	Feasibility study (technical and financial viability)
Document everything formally	Initial risk identification
Gain formal sign-off before proceeding	Stakeholder sign-off document

Key idea

The more precisely requirements are captured here, the less expensive change is later. Every ambiguity in this phase multiplies downstream.

Phase 2 — System and software design

Requirements are allocated to hardware or software subsystems; an overall architecture is established; the fundamental software abstractions and their relationships are identified.

Design sub-activity	What it produces
Architectural design	The overall structure — principal subsystems, their relationships, and how they are distributed across hardware
Interface design	The interfaces between components — how modules communicate, data formats exchanged, API contracts
Component design	Each component's internal design — algorithms, data structures, module logic
Database design	Data structures and their representation — schema, tables, relationships, constraints

The design document becomes the blueprint for the next phase. Programmers should be able to code directly from it without revisiting the requirements document.

Phase 3 — Implementation and unit testing

The design is realised as a set of programs or program units. Each unit (function, class, module) is tested independently against its specification. Defects found here are cheap to fix because no integration has happened yet. Deliverables: compilable source code, unit test cases and results, code review records.

Phase 4 — Integration and system testing

Test level	Purpose
Integration testing	Combine modules and test together; check that interfaces work and data flows correctly between them
System testing	Test the complete integrated system against the original requirements, including performance, security and usability
Acceptance testing	The customer tests with real-world data and formally signs off. The last gate before delivery
Regression testing	After any defect fix, re-check that the fix did not break something that previously worked

Exam pointer

In Waterfall, testing happens *late*. Defects found here are expensive — this is Waterfall's single most-cited weakness and appears in almost every comparison question.

Phase 5 — Operation and maintenance

Normally the longest phase. Errors not discovered earlier are corrected, the implementation is improved, and services are enhanced as new requirements emerge.

Maintenance type	Share	What it involves
Perfective	50%	Improve performance, add features, improve usability from user feedback
Adaptive	25%	Adapt to new hardware, OS updates, changed regulations or APIs
Corrective	20%	Fix defects found by real users in the live system
Preventive	5%	Improve reliability and maintainability before failures occur — refactoring

Exam pointer

Note the counter-intuitive split: *perfective* maintenance — adding and improving features — is the largest category at about half of all maintenance effort. Bug fixing is only a fifth. Maintenance is not mostly about repairing mistakes.

2.4.2 Waterfall — strengths, limitations, and when to use it

Strengths	Limitations
Simple to understand and manage — phases and outputs are clear	Requirements never stand still. Stakeholders change their minds; the assumption that requirements can be fully defined upfront is rarely true
Heavy documentation makes handover easy	Real projects do not follow the sequence. Problems in earlier phases surface constantly; going back means expensive re-entry and re-approval
Easy to measure progress — each phase has a defined output	The customer sees nothing until very late. The first working version appears only at integration and system testing
Works well with fixed-price, fixed-scope contracts	Testing happens at the end. A defect in requirements caught during acceptance testing may require redesigning large parts of the system
Suitable for regulated industries requiring formal sign-off	It assumes uncertainty can be eliminated. Natural early-project uncertainty is treated as if it does not exist, forcing artificial precision

When Waterfall is the right choice

- Requirements are clearly understood and unlikely to change.
- Short projects with small, co-located teams.
- Safety-critical systems — aviation, medical devices, defence.
- Technology and tools are well understood and stable.
- Replacing a known, well-documented existing system.

Real case. NASA space shuttle software — around 4 million lines of code at roughly 0.1 defects per 1,000 lines — used a strict Waterfall process with formal reviews at every phase. The overhead was justified by extreme safety requirements and a stable technical environment.

Key idea

“The fundamental problem of the waterfall model is its emphasis on complete understanding of the problem before solving it — something that is rarely achievable.” (Sommerville)

2.5 Incremental development

Incremental development

Developing an initial implementation, obtaining user feedback, and evolving the software through several versions until the required system is complete. Highest-priority requirements are addressed in the earliest increments.

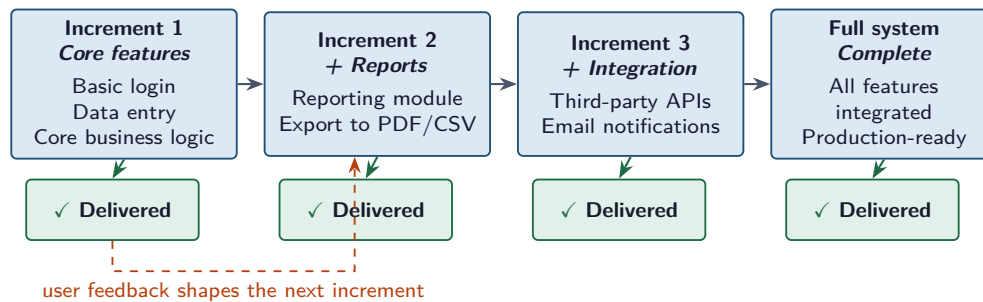


Figure 2.4: Incremental development: each increment adds real, usable functionality and is delivered.

2.5.1 Incremental versus iterative — resolving the confusion

These two terms are used interchangeably in casual speech and mean different things in this subject.

	Incremental development	Iterative development
Type of strategy	A <i>scheduling and staging</i> strategy	A <i>rework scheduling</i> strategy
What happens	The system is divided into parts developed at different times or rates, and integrated as completed	Time is set aside to revisit and improve parts of the system that have already been built
Output	Each increment adds new functionality; typically released to users	The output of an iteration is usually examined internally for modification, not necessarily released
Key question	“What new feature does this increment add?”	“How can we improve what we have already built?”
Alternative to it	‘Big bang’ integration — build everything, then integrate at once (high risk)	Never revisiting a design decision (accumulating debt)

Key idea

Agile combines both: *iterations* improve quality; *increments* add features. A Scrum sprint is simultaneously an iteration (the team refactors and improves) and an increment (the team ships new functionality).

2.5.2 Incremental delivery

Incremental *delivery* goes one step further than incremental *development*: each completed increment is deployed into the customer’s real working environment, not merely handed over internally.

	Step	What happens
1	Define services and priorities	Customers identify which services are most important; this drives the ordering of increments
2	Define increment 1 requirements	Requirements for the first increment are defined in detail; later requirements may still be vague
3	Develop and deploy increment 1	Increment 1 is built, tested and deployed into the real working environment
4	Users experiment and give feedback	Real use clarifies requirements for later increments
5	Define next increment	Requirements for increment 2 are refined, informed by actual experience
6	Repeat until complete	Continue until the system meets all requirements

Exam pointer

One constraint is regularly examined: *once an increment is under development, its requirements are frozen*. Changes go into future increments. This is what makes incremental delivery manageable rather than chaotic.

2.5.3 Benefits and problems

Benefits	Problems
Working software delivered early — customers get value before the full system exists	Process is less visible — managers need documents to measure progress, and frequent small releases produce fewer formal records
Early increments act as prototypes, clarifying requirements for later ones	System structure tends to degrade as increments are added, unless time is deliberately spent refactoring
Lower risk of total project failure — problems in one increment do not kill the project	Hard to replace an existing system — users need all the old functionality before they will switch
Highest-priority features receive the most testing, because they are in the system longest	Difficult to identify the common infrastructure needed by all increments when later requirements are not yet defined
Easier to manage change — new requirements go into future increments	

Key idea

Incremental development is at the heart of most modern approaches — Agile, Scrum and DevOps all build on this idea.

2.6 Verification, validation and the V-model

Verification and validation**Verification**

“Are we building the product right?”

Checks that the software conforms to its specification — internal consistency between what was built and what was specified. Can be done *without* running the software.

Validation

“Are we building the right product?”

Checks that the software meets the real needs of its users. Even a system that perfectly matches its specification is worthless if the specification itself was wrong. Can only be fully done *with* real users.

Verification methods	Validation methods
Code reviews and walkthroughs	Testing with real users
Formal inspections	Acceptance testing with customer data
Static analysis	User feedback and evaluation
Checking against design documents	Prototype evaluation

Key idea

Verification is checking the system against a document. Validation is checking the document against reality.

2.6.1 The V-model

The V-model reorganises Waterfall to make one point visually: every *constructive* phase on the descending arm has a corresponding *verifying* phase on the ascending arm, and the test plans for each level are written when that level is designed — not months later.

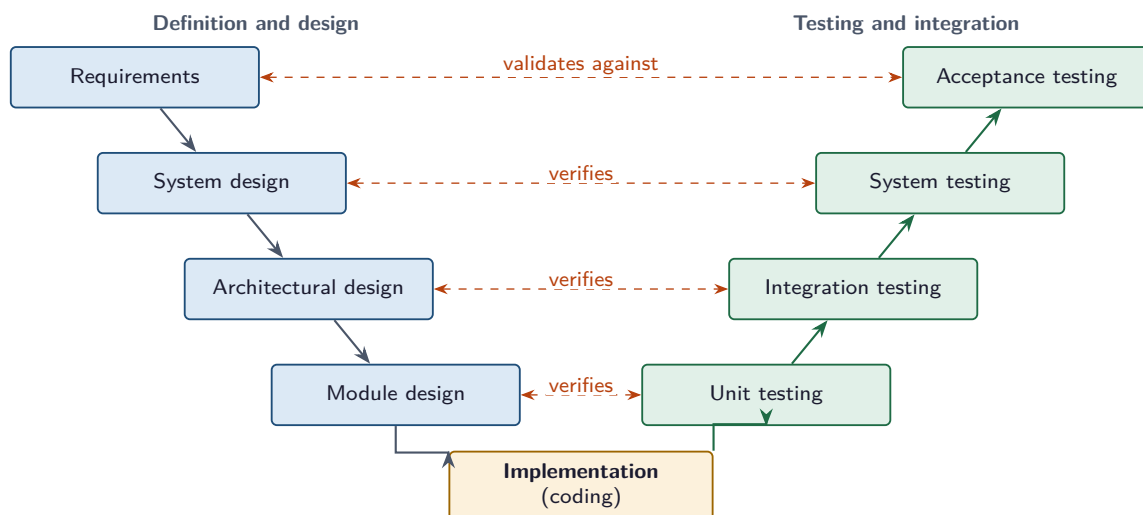


Figure 2.5: The V-model. The dashed links are the point of the diagram: each test level is planned from the artefact directly opposite it.

2.6.2 Stages of testing

Testing occurs at four levels during development, each catching a different class of defect. In Waterfall these happen sequentially near the end; in Agile they are woven throughout.

Stage	Who	Catches
1 Unit / component	Developers	Logic errors, off-by-one bugs, null references, boundary violations
2 Integration	Developers / QA	Interface bugs, data-format mismatches, incorrect API calls, timing issues
3 System	QA team	Missing features, performance failures, security weaknesses, usability problems
4 Acceptance	Customer users	Requirements misunderstandings, missing business rules, usability gaps

Key idea

This is the *process-level* view — where testing sits inside a lifecycle. Chapter 9 is the full treatment: a fifth stage (release testing), how test cases are designed, how tests are written and automated, and how testing is done in practice.

2.6.3 Inspections and reviews — V&V without execution

Testing is not the only way to find defects. Inspections and reviews examine documents and code *without executing* the software, and are often more cost-effective than testing alone.

Technique	Description
Code review	Team members systematically read and examine each other's code. The most commonly used inspection technique. Catches logic errors, style violations, security issues and misunderstandings early.
Formal inspection	A structured meeting where a team examines a document (requirements, design or code) against a checklist. A recorder notes every defect. Studies show inspections catch 60–90% of defects before testing.
Walkthrough	The author presents their work to a group who ask questions and identify issues. Less formal than an inspection; good for early-stage documents where a checklist is not yet appropriate.
Static analysis	Automated tools examine source code without running it, checking for known bug patterns, security vulnerabilities and coding-standard violations. SonarQube, FindBugs, ESLint.

Key idea

Fagan (1976) showed that formal code inspections reduced IBM's testing costs by 25% and delivered 40% fewer defects into system testing. Inspection is the highest-yield quality technique per hour invested.

2.7 Prototyping

Prototype

An initial version of a system, built to demonstrate concepts and try out design options. It is very difficult for end users to anticipate how they will use new software to support their work; if the system is large and complex it is effectively impossible to make that assessment before it is built.

Where helps	prototyping	How
Requirements engineering		Users react to what they can see, revealing requirements they could not articulate in a meeting. See also §4.7.
System design exploration		Prototype competing architectural or interface options to evaluate trade-offs before committing. Especially valuable for user interface design.
Feasibility demonstration		A proof-of-concept prototype shows a technical approach is achievable before the full investment is approved.

Key idea

A prototype must always be evaluated *with user participation*. Building a prototype without involving users defeats its purpose entirely.

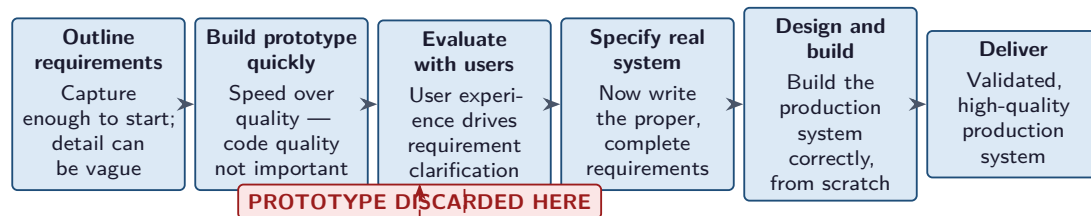
2.7.1 Throwaway prototyping

Figure 2.6: Throwaway prototyping: build to learn, then discard and build properly.

Why discard it?	Benefits
A prototype is built for speed, not quality. It has no error handling, no security, no documentation and no maintainability. Deploying it to production would create long-term technical debt. It has served its purpose: to teach you what to build.	Reduces requirements risk — mistakes are caught early Users react to something visible rather than abstract Better UI/UX design before expensive development begins

2.7.2 Evolutionary prototyping

Here the prototype is *not* discarded. It is continuously refined based on customer feedback until it becomes the delivered product.

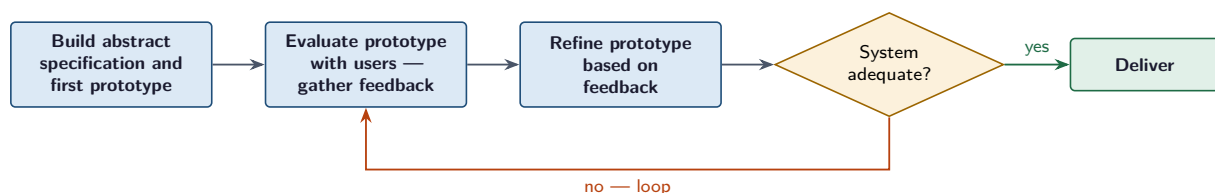


Figure 2.7: Evolutionary prototyping: the prototype becomes the product.

Best suited for	Risks
Systems with rapidly changing or unclear requirements	System structure may degrade without deliberate architectural care
Projects where early delivery of core functionality has high value	Contractual difficulties — hard to fix price and schedule upfront
Smaller systems where full re-engineering each cycle is feasible	The emphasis on speed creates pressure to skip proper documentation

Exam pointer

Compare the two forms directly: **throwaway** prototyping reduces *requirements* risk and starts from the least well-understood requirements; **evolutionary** prototyping delivers a working system and starts from the *best*-understood requirements. Both are “prototyping”, but they solve different problems and are discarded/kept accordingly.

2.8 The Spiral model

Spiral model (Boehm, 1988)

An evolutionary process model that couples the iterative nature of prototyping with the controlled, systematic aspects of the Waterfall model. Software is developed in a series of incremental releases; early iterations may be paper models or prototypes. **Its defining feature is that risk assessment is built into every loop.**

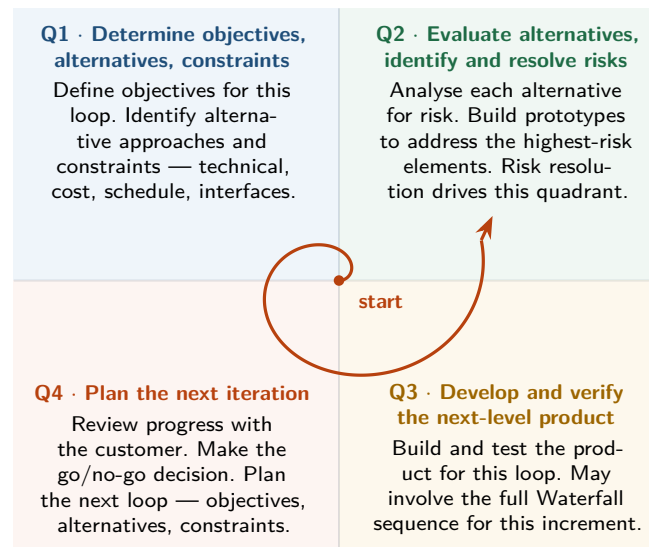


Figure 2.8: The Spiral model. Each complete loop of the spiral is one phase of the project; the radius represents cumulative cost.

2.8.1 Risk as the central driver

In every other plan-driven model risk is acknowledged but not systematically managed. The Spiral model makes risk management the central activity of every phase. The deliverables of each loop — especially early loops — are selected specifically to address the greatest risks first.

Risk type	The question it asks	Typical resolution
Technical	Can we build this at all? Does the technology exist?	Proof-of-concept prototype in loop 1
Requirements	Do we understand what the customer really needs?	Throwaway prototype evaluated with users
Schedule	Can we deliver in the time available?	Realistic planning and milestone tracking
Budget	Will the cost exceed available funding?	Early cost estimation with regular review
Integration	Will the components work together as a whole?	Integration prototyping and interface specification
Strengths		Limitations
Risk management is built into every cycle, not an afterthought		Requires genuine expertise in risk assessment — not all teams have it
Combines structure with flexibility — Waterfall within each loop		Difficult to manage: no fixed phases means more coordination complexity
Prototyping used strategically at every loop as needed		Hard to apply to contracts — customers want fixed price and scope
Customer reviews built in at the end of every loop		Overkill for small projects where risks are already well understood

When to use Spiral

Large and complex systems · high technical or requirements risk · long-duration projects · safety-critical or security-critical systems. The Spiral model's ideas survive in modern risk-driven and DevSecOps approaches, where security risk is addressed continuously rather than at a single gate.

2.9 Rapid Application Development (RAD)

RAD

An incremental process model that emphasises an extremely short development cycle, achieved by a component-based construction approach. If requirements are well understood and project scope is constrained, RAD enables a fully functional system to be delivered in a very short period, typically 60–90 days.

RAD replaces one long build with several parallel team efforts, each producing a major function, which are then integrated.

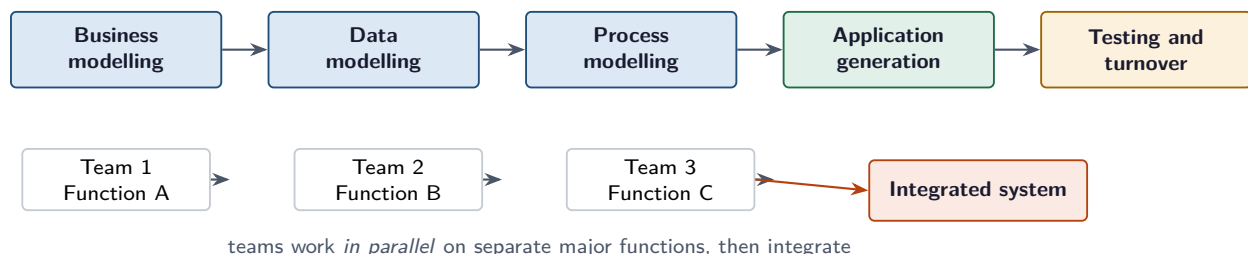


Figure 2.9: RAD: parallel component construction compressing the schedule.

Strengths	Limitations
Very fast delivery when scope is constrained	Requires sufficient human resources to staff parallel teams
Heavy reuse of existing components	Requires developers and customers committed to rapid-fire activity
Strong customer involvement throughout	Not suitable when a system cannot be modularised sensibly
Reduced development time and cost for suitable projects	Not appropriate where high technical risk exists or new technology is being used

2.10 Reuse-oriented software engineering

Reuse-oriented development (CBSE)

An approach that relies on a base of reusable software components and an integrating framework. The system is assembled from existing components rather than developed from scratch. Reuse is now the standard approach for building many types of business system.

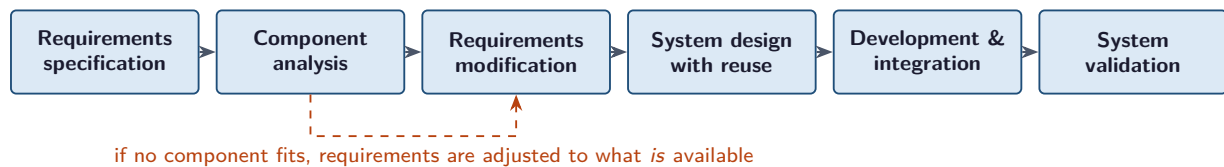


Figure 2.10: The reuse-oriented process. The distinctive step is *requirements modification* — requirements are negotiated against the components that actually exist.

Type of reusable unit	Description
-----------------------	-------------

Web services	Developed to service standards, available for remote invocation over the internet
Object collections	Packages of objects developed as a component or package, integrated with a framework such as .NET or Java EE
Stand-alone systems	Commercial off-the-shelf (COTS) systems configured for a particular environment

Advantages

Reduced cost and risk — less software developed from scratch
Faster delivery

Higher reliability, because reused components are already proven in use

Disadvantages

Requirements compromises are inevitable, so the system may not meet real user needs
Control over system evolution is lost — component vendors set the roadmap
Integration effort can be substantial where component interfaces are incompatible

2.11 The Unified Process

Unified Process (UP / RUP)

A hybrid, use-case-driven, architecture-centric, iterative and incremental process framework. It is described along two axes: four sequential *phases* that structure the project over time, and a set of *disciplines* (workflows) whose effort varies across those phases.

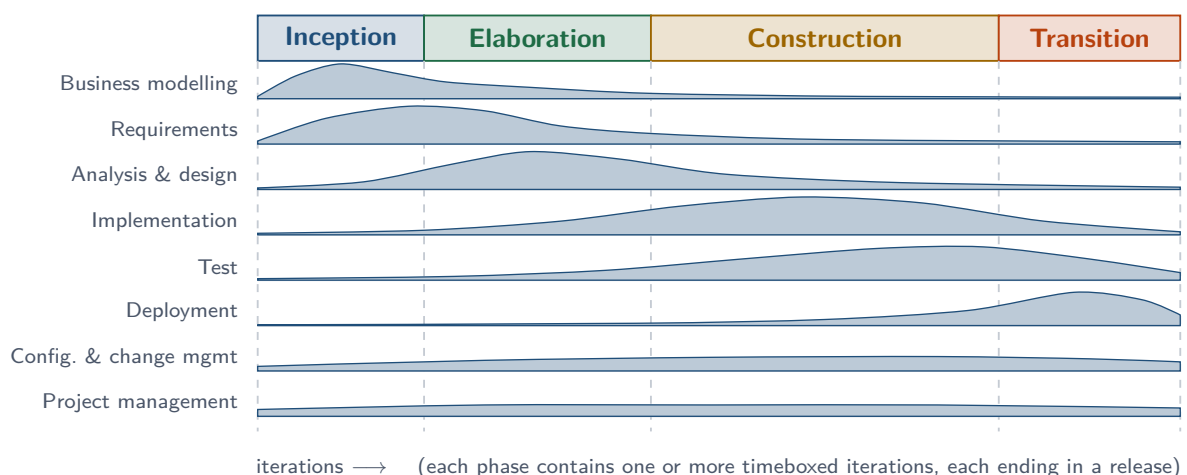


Figure 2.11: The Unified Process “hump chart”. Every discipline runs throughout the project; only the *effort* shifts. This is what makes UP a hybrid — it is phased like a plan-driven model but iterative like an agile one.

Phase	Goal	Milestone
Inception	Establish the business case, scope and rough cost estimate. Is the project worth doing?	Lifecycle Objectives
Elaboration	Capture most requirements, resolve the highest risks, and produce an executable architectural baseline	Lifecycle Architecture
Construction	Build the remaining functionality iteratively on the established architecture	Initial Operational Capability
Transition	Deploy to users: beta testing, training, data migration, tuning	Product Release

UP characteristic	What it means
Use-case driven	Use cases capture functional requirements and drive design, implementation and testing
Architecture-centric	An executable architectural baseline is produced early (end of Elaboration) and everything else builds on it
Iterative and incremental	Each phase contains timeboxed iterations that each produce a working release
Risk-focused	The highest risks are attacked during Elaboration, before the expensive Construction phase begins

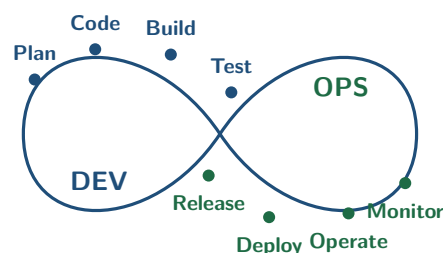
Exam pointer

The most common UP question asks why it is called a *hybrid*. The answer has two halves: it retains plan-driven structure (defined phases, milestones, substantial documentation, formal architecture baseline) while adopting agile mechanics (short iterations, working software each iteration, requirements refined progressively, risk-driven ordering).

2.12 DevOps

DevOps

A culture and set of practices that unite software **D**evelopment and IT **O**perations, so that a change can move from a developer's machine to production quickly, safely and repeatedly. DevOps is the natural extension of incremental delivery: if increments are valuable, then shortening the time to deliver each one is valuable too.



Continuous Integration (CI)

Every commit is automatically built and tested against the

Continuous Delivery (CD)

Every build that passes is automatically made *releasable*.

Continuous Deployment

Every releasable build is automatically pushed to production.

Figure 2.12: The DevOps loop. Monitoring feeds directly back into planning, which is what closes the loop and distinguishes DevOps from simple automation.

DevOps practice	What it achieves
Continuous integration	Developers integrate into the shared mainline several times a day; automated build and test catch integration defects within minutes rather than weeks
Continuous delivery / deployment	The release process is automated and rehearsed constantly, so releasing stops being a risky event
Infrastructure as code	Servers and environments are defined in version-controlled files, so environments are reproducible and drift is eliminated
Automated monitoring and logging	Production behaviour is observable, so problems are detected by the team before users report them
Shared responsibility	The team that builds a service also runs it — “you build it, you run it” — which aligns incentives for reliability
Blameless post-mortems	Failures are analysed for systemic cause, not individual fault, so learning is not suppressed

Exam pointer

DevOps is a *culture*, not a process model. It answers “how do we deliver each increment?” rather than “how do we decide what to build?” Consequently DevOps composes with Scrum, Kanban or even a plan-driven process — it does not replace them.

2.13 Coping with change and process improvement

2.13.1 Why change is inevitable

Change is unavoidable in all large software projects: business changes lead to new and changed requirements; new technologies open new possibilities; and changing platforms require application changes. Change leads to rework, so the cost of change comprises both rework (re-analysing requirements) and the cost of implementing new functionality.

Two complementary strategies reduce that cost:

Strategy	Mechanism
Change avoidance	The process includes activities that anticipate possible changes before significant rework is required — for example, a prototype system is developed and demonstrated to customers so they can refine their requirements before committing high development costs
Change tolerance	The process is designed so that changes can be accommodated at relatively low cost — normally through incremental development, where proposed changes are implemented in increments not yet developed, or by discarding and re-developing a single small increment

2.13.2 Software process improvement

Approach	Focus
Process maturity approach	Improving process and project management, and introducing good software engineering practice. The level of process maturity reflects the extent to which good technical and management practice has been adopted.
Agile approach	Iterative development and reducing overheads in the software process. The primary characteristics are rapid delivery of functionality and responsiveness to changing customer requirements.

The improvement cycle itself is a three-step loop: **measure** one or more attributes of the process or product; **analyse** the current process and identify where quality, schedule or cost bottlenecks arise; **change** by introducing process changes that address the identified weaknesses — then measure again.

The CMMI maturity levels

The Capability Maturity Model Integration describes five levels of process maturity, each of which subsumes the one below it.

1. **Initial** — process is ad hoc; success depends on individual heroics.
2. **Managed** — basic project management: requirements are managed, and projects are planned, performed, measured and controlled.
3. **Defined** — processes are documented and standardised across the organisation.
4. **Quantitatively managed** — process and product quality are measured statistically and controlled.
5. **Optimising** — continuous, measured process improvement is institutionalised.

2.14 AI in software processes

Process activity	Emerging AI assistance
Planning and estimation	Models trained on historical project data suggest effort ranges and flag stories with a high probability of overrun
Development	Code completion and generation inside the IDE; automated refactoring suggestions
Validation	Automated test generation from code or acceptance criteria; test prioritisation based on what changed
Evolution	Automated dependency upgrades, security patch suggestions, and legacy-code explanation
Process measurement	Anomaly detection over delivery metrics — flagging a slowdown before it becomes a missed deadline

Key idea

AI shortens individual activities but does not change the structure of a process. The four fundamental activities — specification, development, validation, evolution — still occur, and the engineer still owns the outcome of each.

2.15 Master comparison of process models

Model	Core idea	Strengths	Weaknesses	Use when
Waterfall	Sequential phases, each signed off before the next	Simple to manage; strong documentation; easy progress measurement; suits fixed-price contracts	No feedback until late; change is expensive; testing at the end; assumes requirements are knowable upfront	Requirements stable; regulated or safety-critical; well-understood technology
Incremental	Deliver the system in successive functional slices	Early value; lower total risk; change absorbed by later increments; priority features tested longest	Less visible to management; structure degrades without refactoring; hard to define shared infrastructure early	Requirements partly known; early delivery has business value
V-model	Waterfall with a matching test level for every design level	Test planning starts early; defect origin is traceable; strong for verification	Same rigidity as Waterfall; no early working software	Safety-critical and regulated systems needing demonstrable test coverage
Prototyping (throwaway)	Build a rough version to learn, then discard it	Resolves requirements ambiguity cheaply; users react to something concrete	Wasted build effort; risk of pressure to ship the prototype	Requirements are unclear or the interface is the main risk
Prototyping (evolutionary)	Grow the prototype into the product	Working system early; continuous user feedback	Structure degrades; hard to price and schedule; documentation suffers	Small systems with rapidly changing requirements
Spiral	Risk-driven loops, each containing a mini-waterfall	Risk managed explicitly every cycle; combines structure with flexibility; customer review each loop	Needs real risk-assessment expertise; complex to manage; hard to contract	Large, complex, high-risk, long-duration projects
RAD	Parallel component construction on a very short cycle	Very fast delivery; heavy reuse; strong customer involvement	Needs many skilled staff; requires a modularisable system; unsuitable for high technical risk	Scope constrained, requirements understood, delivery in 60–90 days
Reuse-oriented (CBSE)	Assemble from existing components	Less code written; faster; higher reliability from proven parts	Requirements compromises; loss of control over evolution; integration effort	Suitable components exist and requirements can flex
Unified Process	Phased, iterative, use-case-driven, architecture-centric	Balances discipline and adaptability; architecture proven early; risk front-loaded	Heavy if applied fully; substantial training and tooling overhead	Medium to large projects needing both governance and adaptability
Agile (Ch. 3)	Short timeboxed cycles, working software, embrace change	Fast feedback; high change tolerance; high customer satisfaction	Less documentation; needs customer availability; scales poorly without extra frameworks	Requirements uncertain or evolving; small co-located empowered teams

Exam pointer

A frequent question format is: “*a company is building X — recommend a process model and justify it.*” Structure the answer around three axes: (1) requirements stability, (2) risk level and criticality, and (3) customer availability and contract type. Name the model, then map each axis to a specific property of that model. Do not simply list advantages.

Agile Software Development

Learning outcomes for this chapter

- LO1.** Explain the Agile Manifesto and its twelve principles in plain language.
- LO2.** Describe Scrum — roles, events, artifacts and the sprint cycle.
- LO3.** Explain XP's key practices — TDD, pair programming and small releases.
- LO4.** Describe how a Kanban board works and what WIP limits achieve.
- LO5.** Explain Lean's origins and its seven principles.
- LO6.** Choose an appropriate agile approach for a given project context.
- LO7.** Identify the main agile tools (Jira, Trello) and how they support teams.
- LO8.** Describe how AI is beginning to change agile practice.

3.1 Why agile?

The plan-driven assumption is that requirements can be fully defined upfront. For most modern software projects that assumption is wrong.

Observation	Consequence
Requirements change constantly	Business needs shift, markets evolve, technology advances — plans made in January are wrong by April
Customers cannot specify upfront	Users do not know exactly what they want until they can see and touch a working system
Speed to market is critical	In a competitive environment, releasing working software in weeks beats planning for a year
Teams need flexibility	Rigid processes create bottlenecks; empowered, self-organising teams respond faster

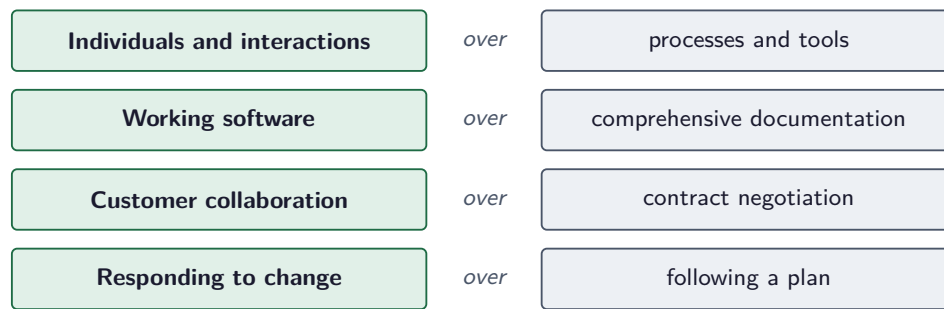
Agile development is based on *learn, adapt, deliver* rather than on rigid process: systems evolve in short iterations, each release is a working system, the focus is on business value, and the design anticipates change. Communication is efficient and progress is visible in real time.

Key idea

Agile is not an excuse for lack of discipline. XP in particular demands *more* engineering discipline than a plan-driven process, not less — test-first development and continuous integration are demanding practices.

3.2 The Agile Manifesto

Seventeen practitioners met at Snowbird, Utah, in 2001 and produced one shared statement.



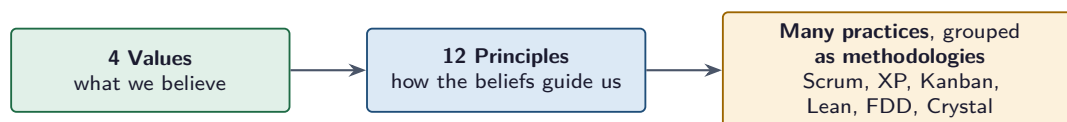
Read the qualifier carefully. “That is, while there is value in the items on the right, we value the items on the left more.” The manifesto does not say documentation, processes, contracts and plans are worthless — a very common misreading, and a very common exam trap.

Figure 3.1: The four values of the Agile Manifesto.

3.2.1 The twelve principles

	Principle	Statement
1	Customer satisfaction	Satisfy the customer through early and continuous delivery of valuable software
2	Welcome change	Welcome changing requirements, even late in development, and use them for the customer’s competitive advantage
3	Deliver frequently	Deliver working software every few weeks; the shorter the timescale, the better
4	Work together daily	Business people and developers must collaborate daily throughout the project
5	Trust motivated individuals	Give them the environment and support they need, and trust them to get the job done
6	Face-to-face conversation	The most efficient and effective method of conveying information within a team
7	Working software = progress	Working software is the primary measure of progress — not documents
8	Sustainable pace	Sponsors, developers and users should be able to maintain a constant pace indefinitely
9	Technical excellence	Continuous attention to good design and engineering enhances agility
10	Simplicity	Maximise the amount of work <i>not</i> done — do only what is needed
11	Self-organising teams	The best architectures, requirements and designs emerge from self-organising teams
12	Reflect and adapt	Regularly reflect on how to become more effective, and adjust accordingly

3.2.2 Values, principles and practices



A team can adopt Scrum’s practices and still not be agile — if it has not adopted the values.
This is the difference between “doing agile” and “being agile”.

Figure 3.2: The relationship between agile values, principles and practices.

3.3 Scrum

Scrum

A lightweight *framework* that helps teams deliver valuable products through short, repeatable cycles called sprints. It is not a process or a technique — it is a framework within which teams can employ various processes and techniques.

Three things make Scrum distinctive:

Time-boxed sprints	All work happens inside fixed 1–4 week cycles. The sprint end date never moves; scope flexes instead.
Inspect and adapt	At every sprint boundary the team examines both what they built and how they worked, and improves both.
Empirical process control	Decisions are based on observation and experimentation, not on a plan made months earlier.

3.3.1 Five values

Commitment	Every team member commits personally to achieving the sprint goal
Focus	Everyone focuses on the sprint work; interruptions are minimised
Openness	The team and stakeholders are open about all work and all challenges
Respect	Team members treat each other as capable, independent professionals
Courage	Team members have the courage to do the right thing, even when it is hard

3.3.2 Three roles

Role	Accountable for	Responsibilities
Product Owner	<i>What</i> to build and in what order	Manages and prioritises the Product Backlog; represents stakeholder and customer needs; accountable for maximising the value of the product; makes decisions on scope, priority and release
Scrum Master	<i>How the team works</i> and improves	Ensures Scrum is understood and followed correctly; facilitates all Scrum events; removes impediments blocking progress; is a coach — not a project manager or line manager
Development Team	<i>How</i> to build the increment	Cross-functional (has all skills needed to deliver); self-organising (nobody tells them how to do the work); 3–9 members; jointly responsible for delivering the Sprint Goal

Exam pointer

These are *accountabilities*, not job titles. The most commonly examined misconception is that the Scrum Master is a project manager. The Scrum Master has no authority over scope, schedule or people; the Product Owner owns scope and priority, and the Development Team owns how the work is done.

3.3.3 Three artifacts

Artifact	Definition	Owned and changed by
Product Backlog	The ordered list of everything that might be needed in the product. It is never complete — it evolves as the product and market change.	Product Owner. Contains user stories, bug fixes, features and improvements, all expressed as value to the customer.
Sprint Backlog	The Product Backlog items selected for the current Sprint, plus the plan for delivering them and achieving the Sprint Goal.	Development Team. Created at Sprint Planning and updated daily; only the team may change it during the Sprint.
Product Increment	The sum of all completed Product Backlog items at the end of a Sprint. Must meet the Definition of Done and be potentially shippable, even if the team chooses not to release it.	The whole Scrum Team. Cumulative — after six sprints the increment contains everything built in all six.

3.3.4 Five events

	Event	Time-box	Purpose
1	Sprint	1–4 weeks	The container for all other events. A time-box within which the team delivers a potentially shippable increment
2	Sprint Planning	Max 8 hours	The team collaborates to define the Sprint Goal and select the backlog items it will deliver
3	Daily Scrum	15 minutes	Daily synchronisation. Three questions: what did I do yesterday, what will I do today, what is blocking me
4	Sprint Review	Max 4 hours	The team demonstrates the increment to stakeholders and collects feedback to update the Product Backlog
5	Sprint Retrospective	Max 3 hours	The team reflects on how it worked and identifies one or two concrete improvements for the next Sprint

Exam pointer

Time-boxes are maxima for a *one-month* sprint and scale down proportionally — an event ends when its purpose is achieved. Note also the difference between the Review and the Retrospective: the **Review** inspects the *product* with stakeholders; the **Retrospective** inspects the *process* within the team. Swapping these is the single most common Scrum error in exam answers.

3.3.5 The sprint cycle

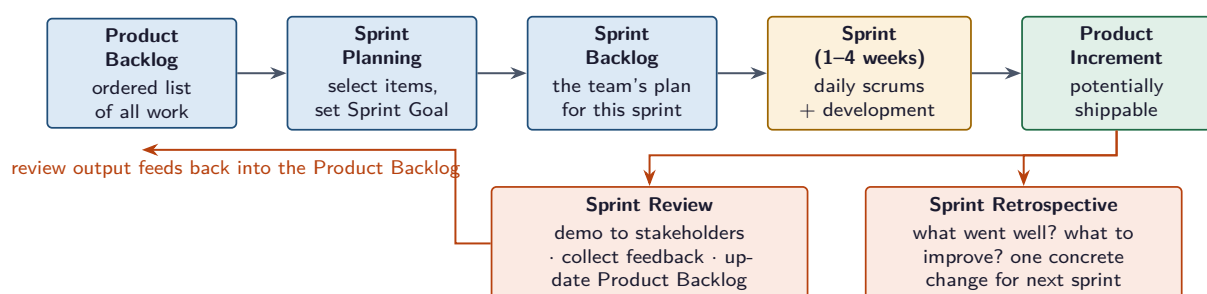


Figure 3.3: The Scrum sprint cycle. Every sprint follows the same repeating pattern: inspect, adapt, deliver, repeat.

3.3.6 The Definition of Done

Definition of Done (DoD)

A shared understanding of what “complete” means. A Product Backlog item is not done until it meets *every* criterion in the DoD.

Why the Definition of Done matters

It creates transparency — everyone knows exactly what “done” looks like

It prevents undone work from accumulating as hidden technical debt

Without it, different team members apply different standards, leading to surprises at Sprint Review

It evolves as the team improves — a stricter DoD means higher quality

A typical Definition of Done

Code written · unit tests written and passing · code reviewed by a peer · integrated into the main branch · deployed to the test environment · acceptance criteria verified · documentation updated.

3.3.7 Tracking a sprint — the burndown chart

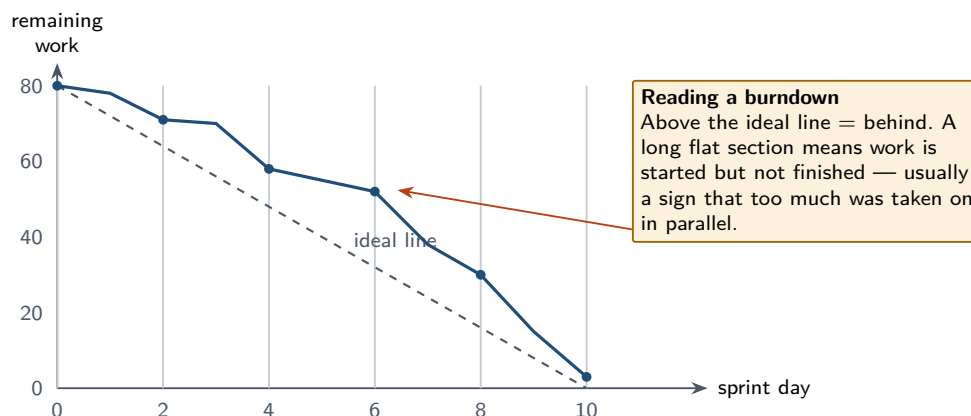


Figure 3.4: A sprint burndown chart. The team is behind mid-sprint and recovers late — a common and diagnosable pattern.

3.3.8 Strengths and challenges

Strengths

Delivers working software frequently, building stakeholder trust
 Continuous feedback reduces the risk of building the wrong thing
 Simple to understand; teams can be trained quickly
 Retrospectives create a culture of continuous improvement
 Works well for complex product development where requirements change

Challenges

Requires genuine cultural change — not just renaming roles and scheduling meetings
 The Product Owner must be available and empowered, which is often a struggle in practice
 The Scrum Master role is frequently misunderstood as project management
 Does not scale naturally to very large or distributed teams
 Produces less documentation, which can be a problem in regulated industries

3.4 Extreme Programming

Extreme Programming (Kent Beck, 1999)

An agile methodology focused on engineering excellence and high-quality software delivered in frequent, short releases. It takes proven software development practices to their extreme: *if something works, do it as much as possible.*

If this works...	...then do it to the extreme
Testing works	Write tests <i>before</i> you write code (TDD)
Code review works	Review code continuously, by pairing
Design works	Refactor and redesign constantly, not only at the start
Simplicity is good	Always do the simplest thing that could possibly work
Short iterations work	Deliver new releases to customers every 1–3 weeks
Customer feedback works	Put a real customer in the team, full time

3.4.1 Five values

Communication	Constant face-to-face interaction — the team talks first, documents second
Simplicity	Always do the simplest thing that works; avoid complexity until it is necessary
Feedback	Fast, frequent feedback from tests, customer and team — the shorter the loop the better
Courage	Tell the truth about progress, throw away bad code, accept and adapt to late changes
Respect	Every team member's contribution matters; mutual respect enables collective ownership

3.4.2 The twelve practices

Programming practices

Test-Driven Development	Write the test before the code. Red → Green → Refactor. Tests serve as executable specification.
Pair programming	Two developers, one machine. The <i>driver</i> writes; the <i>navigator</i> reviews and plans ahead. Roles switch regularly.
Refactoring	Continuously improve the internal structure of the code without changing its external behaviour.
Simple design	Do only what is needed now. No speculative features. YAGNI — “You Ain’t Gonna Need It”.

Team practices

Collective code ownership	Anyone may improve any part of the codebase at any time. No individual owns a module.
Continuous integration	Integrate and build multiple times per day. Never let integration debt accumulate.
Coding standards	Shared standards enable collective ownership and refactoring without friction.
Sustainable pace	The 40-hour week. No sustained overtime — tired engineers make mistakes, so pace is a quality practice.
Metaphor	A shared system description giving common vocabulary and an architectural vision understood by all.

Planning and feedback practices

On-site customer	A real customer embedded in the team — writes stories, runs acceptance tests, answers questions daily.
Planning game	The customer defines business value through stories; developers estimate effort. Release planning is negotiated between the two.
Small releases	Release to production frequently — weeks, not months. Real feedback from real users.

3.4.3 Test-driven development

In TDD you never write production code unless you have a failing test first. Each cycle takes only a few minutes.

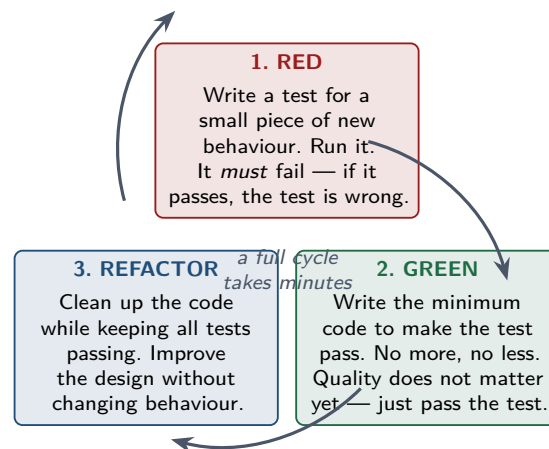


Figure 3.5: The TDD cycle. TDD does not eliminate defects — it catches them early, when fixing them costs almost nothing.

3.4.4 User stories and INVEST

Both XP and Scrum use user stories as the standard way to express requirements.

User story format

As a [type of user], I want to [do something], so that [I achieve some goal].

Example. *Title:* User Login. “As a registered user, I want to log in with my email and password, so that I can access my personal dashboard.”

A good user story is **INVEST**:

I	Independent	Stories stand alone — not dependent on another story
N	Negotiable	A reminder for a conversation, not a fixed contract
V	Valuable	Delivers value to the end user or customer
E	Estimable	The team can estimate the effort to complete it
S	Small	Completable within one sprint, usually a few days
T	Testable	Has clear acceptance criteria that can be verified

Exam pointer

A story with no acceptance criteria is not testable and therefore fails INVEST. When asked to “write a user story”, always supply both the story sentence *and* two or three acceptance criteria — that is what distinguishes a full-mark answer.

3.4.5 Strengths and fit

Strengths	Best suited when
High code quality through TDD and constant review	Small (2–10 person), technically strong team
Low defect rates — bugs caught immediately by tests	Customer available for frequent interaction
Team flexibility through collective ownership	Requirements evolve rapidly
Sustainable pace protects long-term productivity	Engineering quality is the top priority

Key idea

Scrum is a *management* framework — it organises who does what and when. XP is an *engineering* discipline — it specifies how the code is written. They are complementary, and many teams combine Scrum’s sprint structure with XP’s technical practices.

3.5 Kanban

Kanban

A method for visualising work, limiting how much work is in progress, and managing the flow of value from request to delivery. Unlike Scrum it has no fixed sprints, no prescribed roles and no prescribed ceremonies.

The word *kanban* is Japanese for “sign-board” or “card”. It comes from Toyota, which used physical cards from the 1940s to signal when to replenish parts on a production line. David Anderson adapted the idea for software development in 2007.

Core property	Meaning
Visualise the workflow	All work is made visible on a board — nothing is hidden or assumed
Limit work-in-progress	Constraints on how many items may be in progress at once, driving focus and flow
Manage and improve flow	Measure and continuously improve how fast work moves from start to done

3.5.1 The Kanban board

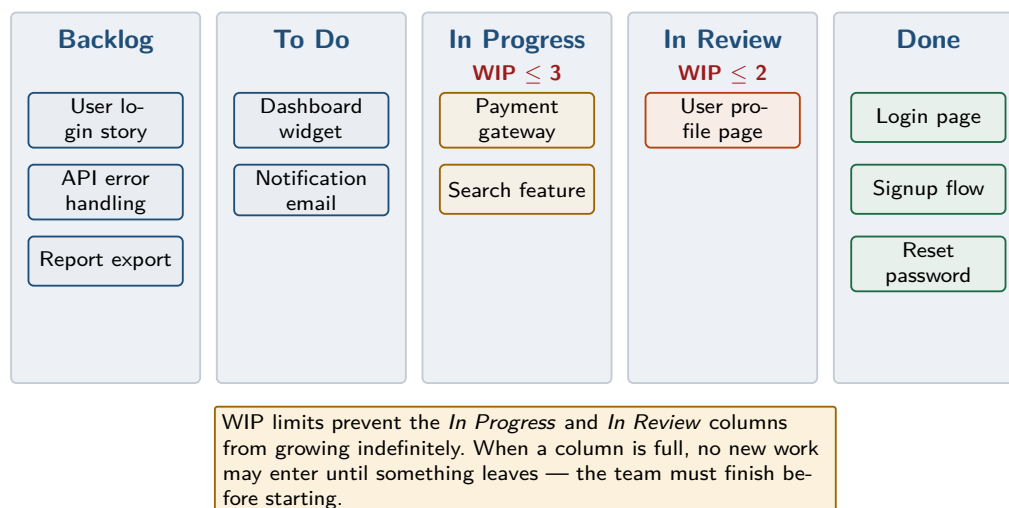


Figure 3.6: A Kanban board. Cards move left to right as work progresses.

3.5.2 Work-in-progress limits

WIP limit

A maximum number of work items allowed in a workflow stage at any one time. When a column is full, no new work can enter until something exits.

Without WIP limits	With WIP limits
People start many things but finish few	Bottlenecks become visible immediately
Defects are discovered late	The team swarms to unblock stalled items
Context-switching destroys productivity	Less context-switching, so less waste
The team looks busy but delivers slowly	Faster flow from start to done

Key idea

“Stop starting. Start finishing.” — the golden rule of Kanban. Limiting work in progress *increases* throughput, which is the most counter-intuitive result in the subject.

3.5.3 Six core practices

1	Visualise the workflow	Make all work visible. Map every step from customer request to delivery. Nothing hidden in an inbox or someone’s head.
2	Limit WIP	Set and enforce WIP limits per stage. When a column is full, stop starting and help finish.
3	Manage flow	Monitor how work moves through the system. Identify and remove bottlenecks. Track cycle time and throughput.
4	Make policies explicit	Write down the rules for how work moves — what “ready” and “done” mean. Visible policies reduce arguments.
5	Implement feedback loops	Daily stand-ups, weekly replenishment meetings and monthly service-delivery reviews.
6	Improve collaboratively	Use data — cycle time, throughput, cumulative flow — to make small, evidence-based, evolutionary changes.

3.5.4 Scrum versus Kanban

Factor	Scrum	Kanban
Cadence	Fixed sprints (1–4 weeks)	Continuous flow — no sprints
Roles	Scrum Master, Product Owner, Development Team	No prescribed roles
Planning	Sprint Planning every sprint	On demand — replenish when capacity exists
WIP limits	Implicit (sprint capacity)	Explicit — enforced per column
Change	No changes mid-sprint	New items can be added at any time
Output metric	Velocity (story points per sprint)	Cycle time and throughput
Best for	Product development with clear sprints	Continuous service, support and maintenance

Key idea

Many teams combine both: Scrum for new feature development, Kanban for bug fixes and support work. The hybrid is often called “Scrumban”.

3.6 Lean software development

Lean did not start in software. The Toyota Production System, developed from the 1950s to the 1970s, revolutionised manufacturing by eliminating every form of waste. Mary and Tom Poppendieck adapted these principles for software in their 2003 book *Lean Software Development*.

Key idea

Identify and deliver value to the customer. Identify every step in the value stream. Eliminate every step that does not add value. Then make the value-adding steps flow without interruption.

The Agile Manifesto was directly influenced by Lean thinking; Kanban is essentially a Lean technique applied to software workflow; DevOps adopted Lean's value-stream mapping as a core practice; and many agile principles — simplicity, flow, sustainable pace — come directly from Lean.

3.6.1 Seven principles

1	Eliminate waste	Remove anything that does not add value — partially done work, extra features, waiting, unnecessary process
2	Amplify learning	Use short feedback cycles and experimentation to accumulate knowledge quickly
3	Decide as late as possible	Delay irreversible decisions until the last responsible moment, when you will have more information
4	Deliver as fast as possible	Short cycle times reduce waste, increase feedback and lower business risk
5	Empower the team	Decisions are made by those doing the work — the team closest to the problem has the best knowledge
6	Build integrity in	Quality is not tested in at the end; it is built in from the start through good design
7	See the whole	Optimise the entire system, not individual parts — local optimisation often creates global waste

3.6.2 Seven wastes in software

1	Partially done work	Code that is 90% complete and sitting in a branch for weeks
2	Extra features	Building features nobody asked for — “it might be useful someday”
3	Re-learning	Solving the same problem again because knowledge was not captured
4	Handoffs	Passing work between teams introduces delay and miscommunication
5	Task switching	Moving between multiple work items at once — all progress slows
6	Delays and waiting	Waiting for approvals, environments, test results or another team
7	Defects	Bugs — especially those found late, when the cost of fixing is highest

3.7 Other agile methods

Method	Description
Crystal (Cockburn)	A <i>family</i> of methods that vary by team size and project criticality: Crystal Clear (1–6 people), Yellow (7–20), Orange (21–40), Red (41–80). All share a focus on people, interaction, community, skills and communication rather than heavy process. Use for small to medium teams wanting a lightweight, people-centric approach.
Feature-Driven Development (De Luca & Coad)	Work is planned and tracked by individual features. Five processes: develop an overall model · build a feature list · plan by feature · design by feature · build by feature. Features follow the form <i>action + result + object</i> , e.g. “calculate the total of a sale”. Use for larger teams (up to 250) in enterprise settings where feature tracking matters.
DSDM	Dynamic Systems Development Method. Fixes time, cost and quality, and flexes scope, using MoSCoW prioritisation (§4.9). One of the earliest agile frameworks and unusually strong on governance.

3.8 Choosing an agile approach

Method	Best for	Key strength	Key challenge	Team size
Scrum	Product development	Structure plus feedback loops	Role confusion; requires cultural change	3–9
XP	High-quality engineering	TDD, pairing, CI quality	Requires strong technical discipline	2–10
Kanban	Service and maintenance	Flow plus WIP limits	No cadence means less predictability	Any
Lean	Waste-reduction mindset	Whole-system thinking	Needs experienced practitioners	Any
Crystal	People-focused projects	Flexible and lightweight	Less guidance for new teams	1–80
FDD	Large enterprise	Feature-level tracking	More upfront planning needed	Up to 250

If the project context is...	... then consider
Requirements unclear or evolving rapidly	Scrum or XP — frequent delivery plus customer feedback
Continuous delivery needed with no fixed cadence	Kanban — manage flow, no sprints required
Engineering quality is the absolute priority	XP — TDD, pair programming, continuous integration
Waste elimination across the whole process	Lean thinking combined with Kanban
Small team wanting minimal process overhead	Crystal Clear
Large enterprise tracking hundreds of features	FDD, or Scrum at scale (SAFe, LeSS)

3.8.1 Agile at scale

Scrum and Kanban work well for teams of 3–15 people. Enterprises with hundreds of developers on one product need additional coordination frameworks.

Framework	Approach	Use when
SAFe	Scaled Agile Framework. The most widely used. Organises teams into Agile Release Trains (ARTs); five levels from team to portfolio	Large enterprises with many teams and complex governance requirements
LeSS	Large-Scale Scrum. Scrum applied across multiple teams with minimal process overhead: one Product Backlog, one Product Owner, many Scrum teams	2–8 teams on the same product, preferring lightweight coordination
Nexus	Created by Scrum.org. Extends Scrum for 3–9 teams with a Nexus Integration Team managing dependencies	30–80 developers on a single product with strong integration challenges

3.9 Agile tools

A team practising Scrum or Kanban needs tools that make work visible and coordination easy. Agile tools replace the physical wall of sticky notes with a digital board everyone can see, whether co-located or distributed.

Category	Representative tools
Project and backlog management	Jira, Azure DevOps, Linear, Monday.com
Visual task boards	Trello, Miro, Asana, ClickUp
Collaboration and communication	Confluence, Notion, Slack, Teams
Reporting and metrics	Jira burndown, GitHub Insights, velocity charts

3.9.1 Jira

Developed by Atlassian and used by more than 65,000 organisations, Jira supports both Scrum and Kanban workflows out of the box.

Feature	What it does
Product backlog	Create and manage stories, bugs and tasks; assign story points, labels and priority. The Product Owner prioritises the backlog in one view.
Sprint board	Shows all items in the current sprint: To Do, In Progress, Done. Team members drag cards as work progresses.
Kanban board	Configure columns to match your workflow; set WIP limits per column, which Jira enforces automatically.
Burndown charts	Auto-generated charts of remaining work against time, visible to the whole team and stakeholders.
Velocity reports	Track story points completed per sprint; used to forecast future sprint capacity.

Jira issue hierarchy — the practical workflow

1. **Create the project.** Choose a *team-managed* project with a Scrum template (for sprints) or a Kanban template (for continuous flow).
2. **Understand the sections.** *Backlog* — all tasks and stories; *Board* — drag-and-drop task management; *Roadmap* — timeline planning; *Reports* — burndown and velocity (Scrum only).
3. **Create issues** of the right type:
 - **Epic** — a large feature area, e.g. “User Management”
 - **Story** — a piece of user-facing functionality, e.g. “User Login”
 - **Task** — a unit of work, e.g. “Design the login UI”
 - **Bug** — something to fix
4. **Fill in the details:** summary, description, assignee, priority, labels.
5. **Link issues:** stories to epics, bugs to their related stories.
6. **Manage the backlog:** add user stories, break large items into sub-tasks, order by priority.
7. **Run the sprint:** plan it, move cards across the board, review the burndown, close the sprint.

3.9.2 Trello

Also from Atlassian, Trello is a pure Kanban tool ideal for teams learning the method. Work is organised as *boards* → *lists* → *cards*.

A Trello board structured for a student project

A useful six-list structure that maps a Scrum-like flow onto a Kanban board:



Each card carries a **description**, an assigned **member**, a **due date**, a **checklist** breaking the task into steps, and a colour **label** (for example red for a feature, yellow for a bug fix). Comments record what was done at each move, and @**mentions** notify teammates.

Automation. Trello *Rules* remove repetitive work. A typical first rule: *when a card is added to the list “Sprint Backlog” by anyone, copy the card to the top of the list “To Do”*. A second common rule: *when a card is moved to “Done”, post a comment “This task has been*

completed”. Automation is worth understanding because it is a small, concrete instance of the DevOps principle that repeated manual steps should be mechanised.

Tracking. The board’s *Activity* log records every card movement, comment and assignment — the audit trail that substitutes for formal status reporting.

Key idea

Trello is ideal for *learning* Kanban — the board is the method, made visible. Jira is where most professional teams live, because it adds backlog hierarchy, estimation, sprint mechanics and reporting.

3.9.3 Other tools worth knowing

Tool	Vendor	Character
Azure DevOps	Microsoft	Full DevOps pipeline — boards, repos, pipelines, test plans; deep Microsoft-ecosystem integration
Linear	Linear Inc.	Modern, fast issue tracker popular with startups; clean, keyboard-first design
Monday.com	Monday	Flexible work-management platform, also used outside software in marketing and HR
GitHub Projects	GitHub	Task board integrated directly into GitHub; ideal for open-source and GitHub-centric teams
Miro	Miro	Virtual whiteboard for retrospectives, planning poker, roadmapping and brainstorming

3.10 AI in agile development

AI tools are being integrated into every stage of the agile cycle. The engineer’s role shifts from writing every line to directing, reviewing and validating AI-generated work.

Before	After
Writing boilerplate code	Reviewing, testing and improving AI-generated code
Manual test case creation	Verifying and refining AI-generated test suites
Estimating story effort unaided	Using AI predictions as a starting point for discussion
Writing meeting summaries	Editing AI-drafted summaries of stand-ups and retrospectives

Agile activity	How AI assists
Story refinement	Reads raw requirements and suggests story breakdowns, acceptance criteria and edge cases the team may not have considered
Effort estimation	Models trained on historical project data predict story-point ranges from similar past work, giving a data-informed starting point
Dependency detection	Scans the backlog and flags stories depending on other stories or external systems, preventing blocked sprints
Risk identification	Analyses technical complexity and history to flag stories with the highest risk of overrun or defects
Capacity suggestion	Combines team velocity, planned absences and historical patterns to suggest a realistic sprint capacity
Coding and review	Copilot, Cursor, Claude and ChatGPT suggest functions, explain unfamiliar code, propose refactorings and draft tests
Testing	Automated test generation; test prioritisation based on what changed; bug triage and severity prediction
Retrospectives	Real-time transcription and summarisation; sentiment analysis across sprints; automatic extraction of action items into tickets

Key idea

AI in retrospectives is most valuable when it surfaces patterns *across many sprints* — patterns a team may not notice sprint by sprint. That is a genuinely new capability, not simply a faster version of something humans already did.

Exam pointer

Whatever the activity, the examinable point is constant: AI-generated work must still be read, understood, tested and owned by the engineer. Professional responsibility does not transfer to the tool.

Requirements Engineering

Learning outcomes for this chapter

- LO1.** Explain what a software requirement is and describe the three dimensions — functional, non-functional and domain.
- LO2.** Explain why ambiguous requirements are dangerous and identify the quality properties of a good requirement.
- LO3.** Describe the RE process and how it differs in plan-driven and agile contexts.
- LO4.** Identify common elicitation techniques and explain how requirements are discovered.
- LO5.** Explain what a Software Requirements Specification is and how it compares to an agile product backlog.
- LO6.** Describe how requirements are validated and managed throughout the lifecycle.
- LO7.** Describe how AI is beginning to assist requirements engineering.

4.1 Why requirements matter

A defect found during requirements costs almost nothing to fix. The same defect found after deployment costs roughly a hundred times more, because rework, redesign, retesting and redeployment all compound.

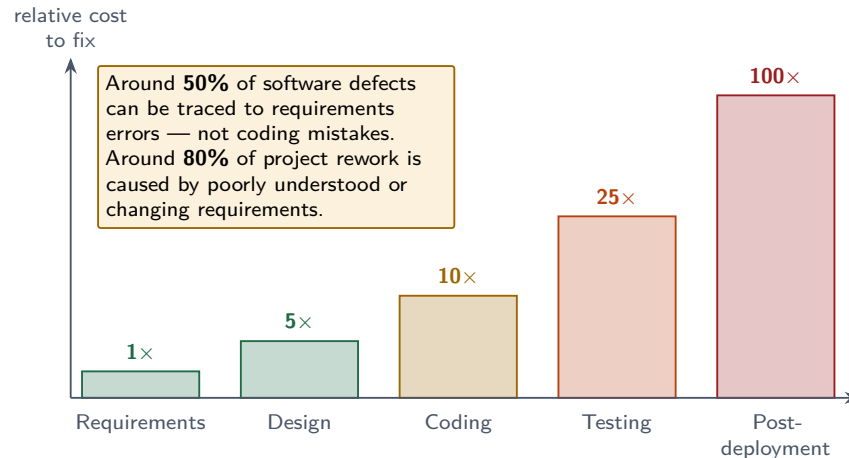


Figure 4.1: The cost of fixing a defect, by the stage at which it is discovered.

4.2 Requirements engineering in both cultures

Aspect	Plan-driven	Agile
How requirements look	Numbered sentences in a formal SRS document	User stories on a backlog: “As a . . . , I want . . . , so that . . . ”
When they are written	Before development begins — all upfront	Just before they are needed — incrementally
How detailed	Complete, precise, formal, with measurements	Lightweight — enough to have a conversation
How change is handled	Formal change request, reviewed and approved	Reprioritise the backlog — quick and continuous
Main document	Software Requirements Specification (SRS)	Product Backlog with acceptance criteria
Best when	Requirements stable; regulated or contractual	Requirements uncertain; customer needs evolve frequently

Key idea

Most projects today use a mixture. The plan-driven techniques are not obsolete — they are the tools you reach for when the cost of getting it wrong is high and the cost of changing later is prohibitive.

4.3 What is a requirement?

Requirement

A condition or capability that must be satisfied — needed by a user to achieve a goal, or required of a system to meet a standard or constraint.

A requirement can take three forms, and confusing them causes real problems:

Form	What it is	Example
User need	The goal, expressed in the user’s own language — what the person wants to achieve	“I want to track my daily fitness.”
System rule	The capability the system must provide in order to meet that need	“The system shall record daily step counts and display weekly charts.”
Documented form	The artefact that captures, communicates and preserves the requirement	A user story card: “As a user, I want to track my steps so that I can stay healthy.”

4.3.1 The lifecycle of a requirement

A requirement does not spring fully formed from a meeting; it evolves.

1	Need identified	A stakeholder expresses a goal, problem or desire — a user, manager or regulator
2	Problem understood	The gap between the current situation and the desired outcome is analysed. Why does this need exist?
3	Requirement defined	The need is formalised into a clear, actionable requirement, precise enough to design and test
4	Solution proposed	A system feature or change is designed to fulfil it. One requirement may have several design options
5	Conflicts managed	A requirement may trigger others, or conflict with existing ones. Conflicts must be resolved
6	Prioritised	Not everything can be built. Decisions are made about what to include, defer or drop

4.4 The three dimensions of requirements

Every requirement answers one of three questions: *what*, *how well*, or *why*.

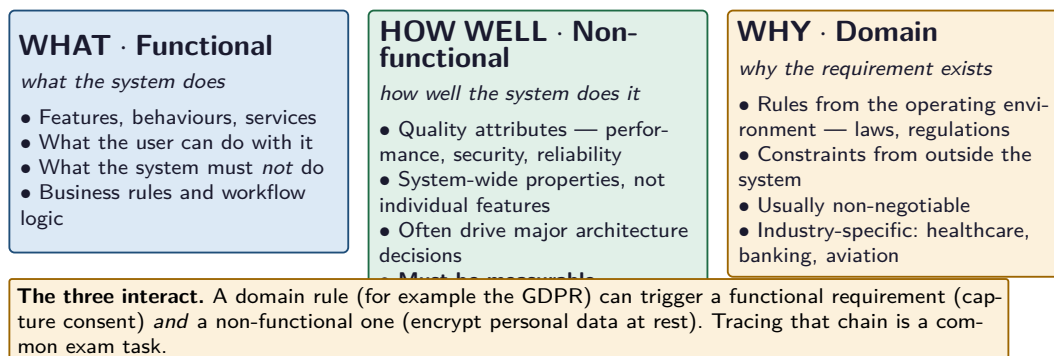


Figure 4.2: The three dimensions of a requirement.

4.4.1 Functional requirements

Functional requirement

Describes the behaviours, services and features of the system — what it does, what the user can do with it, and what it must never do.

System	Example functional requirement
Online banking	The system shall allow a registered user to transfer funds to another account
E-commerce	The system shall apply a 10% discount when the basket total exceeds LKR 5,000
University portal	The system shall generate and email a student's exam results within 2 hours of results being published
Fitness app	The system shall record the user's daily step count and display a weekly bar chart

4.4.2 Non-functional requirements

Non-functional requirement

Describes the quality attributes of the system — how reliably it runs, how fast it responds, how secure it is, how easy it is to use. NFRs are system-wide, not tied to individual features.

Category	Example
Performance	The system shall process a search query and return results in under 1 second for 95% of requests
Reliability	The system shall be available 99.9% of the time — no more than 8 hours of downtime per year
Security	Only registered administrators shall be able to modify student grade records
Usability	A new user shall be able to complete account registration in under 3 minutes with no training
Maintainability	A developer shall be able to add a new payment method without modifying existing payment code
Interoperability	The system shall accept patient data formatted using the HL7 FHIR standard

Bad (vague)	Good (measurable)
✗ The system shall be fast.	✓ The system shall respond to any search query in under 1 second for 95% of requests under normal load.
✗ The system shall be user-friendly.	✓ A first-time user shall complete the registration form in under 3 minutes without requesting help.
✗ The system shall be highly available.	✓ The system shall be available 99.9% of the time, measured over any 30-day period.
✗ The system shall be secure.	✓ All patient data shall be encrypted using AES-256, both at rest and in transit.

Key idea

If you cannot write a test for it, it is not a requirement. A vague non-functional requirement is not a requirement — it is a wish.

4.4.3 Domain requirements

Domain requirement

A constraint arising from the industry, legal regulation or operating environment. Domain requirements explain *why* certain behaviours are necessary, beyond what any individual user personally wants.

Domain	Domain rule	Triggers FR	Triggers NFR
Healthcare	Patient records must be retained for 10 years	System shall archive records with a 10-year retention policy	Storage must maintain 99.9% data integrity over the retention period
Banking	All transactions must comply with Anti-Money-Laundering law	System shall flag transactions above LKR 1 million for compliance review	Audit logs must be immutable and retained for 5 years
Education	Grades must follow the government-defined grading scale	System shall calculate GPA using the national 4.0 standard only	GPA calculations must be independently auditable by the Ministry

The three dimensions in one system — a hospital appointment app

Functional

- The system shall let a patient book an appointment with an available doctor.
- The system shall send a reminder SMS 24 hours before an appointment.

Non-functional

- The booking page shall load in under 2 seconds for 95% of requests.
- The system shall be available 99.9% of the time, including weekends and public holidays.

Domain

- Patient medical records shall be retained for 7 years (Health Ministry regulation, Sri Lanka).
- Prescriptions shall only be issued by practitioners registered with the SLMC.

4.5 Requirement quality

4.5.1 The danger of ambiguity

One sentence, two systems

Ambiguous requirement: “The system shall allow users to search for patients.”

Developer A reads it as: search by name within the currently selected clinic only — a small, simple feature taking two days.

Developer B reads it as: search by name across *all* clinics and *all* appointments — a large, complex feature taking three weeks.

Two developers build two different systems, neither of which is what the customer wanted.

4.5.2 Sharpening a requirement

Version 1 (original)	“The system shall allow users to search for patients.” <i>Who is searching? By what field? In which clinic? Completely ambiguous.</i>
Version 2 (better)	“The system shall allow clinic staff to search for a patient by full name within the currently selected clinic.” <i>Better — we now know who, what field and what scope. But what if two patients share a name?</i>
Version 3 (good)	“The system shall allow clinic staff to search for a patient by full name within the currently selected clinic. If multiple patients share the same name, results shall display the patient’s date of birth and clinic ID to distinguish them.” <i>Clear, complete and testable. A developer can build it; a tester can verify it.</i>

4.5.3 Properties of a good requirement

Clear	Only one reasonable interpretation is possible. Every reader — engineer, tester, customer — understands it identically
Complete	Nothing essential is missing. The requirement says everything needed to design and test the feature
Consistent	It does not contradict any other requirement in the document
Verifiable	You can write a test for it. If you cannot think how to test it, it is too vague
Traceable	Linked backward to a stakeholder need and forward to a design decision or test case — you know <i>why</i> it exists
Feasible	Technically possible within the available budget, time and technology

4.6 The requirements engineering process

Requirements Engineering

The systematic and disciplined process of discovering, analysing, documenting, validating and managing the requirements for a software system. It covers *what* the system must do, *how well* it must do it, and *why* certain behaviours are needed.

Why is it called “engineering”? Because it follows repeatable steps rather than ad hoc collection in a single meeting; it is rigorous about ambiguity, traceability and completeness; it resolves conflicts between stakeholders within real constraints; it uses models, templates and formal notations; and it is iterative by nature, adapting as understanding evolves.

4.6.1 Feasibility study

A short, focused assessment carried out at the very beginning to determine whether the proposed system is realistic, financially justified and aligned with the organisation’s goals.

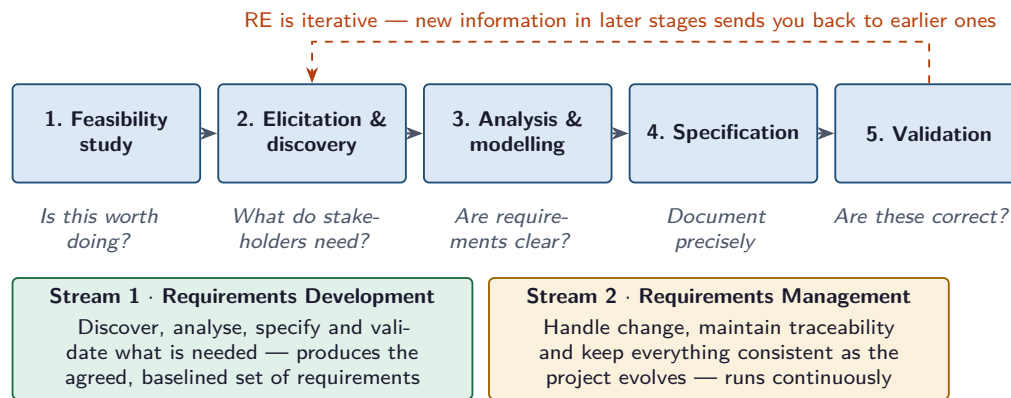


Figure 4.3: The requirements engineering process: five activities forming a cycle, plus two parallel streams that run for the life of the project.

Type	The question it answers
Technical	Can it be built with available technology and skills? Are there technical risks that could stop the project?
Economic	Do the benefits justify the cost? Is there a clear return on investment? Is the budget sufficient?
Operational	Will the organisation actually use it? Does it fit how people work? Will users accept and adopt it?
Legal	Does it comply with relevant laws, data-protection regulations, licences and contractual obligations?
Schedule	Can it be built within the required timeframe? Are there external deadlines that must be met?
Resource	Are the necessary personnel, infrastructure and tools available?
Cultural	Will the system align with user and organisational culture?
Financial	Will funding and cash flow sustain development to completion?

Exam pointer

The first five — **T**echnical, **E**conomic, **L**egal, **O**perational, **S**chedule — are sometimes remembered as *TELOS* and are the set most commonly required in an answer. The output of a feasibility study is a recommendation to *proceed*, *modify scope*, or *abandon* — before any significant investment is made.

4.7 Requirements elicitation

Key idea

Elicitation is not “requirements gathering”. Requirements are rarely lying around ready to be collected — they must be actively drawn out, discovered, and often co-created through conversation with stakeholders.

4.7.1 Why elicitation is difficult

- Stakeholders often do not know exactly what they want until they see a working version.
- Different stakeholders have different — and sometimes conflicting — needs and priorities.
- People describe imagined *solutions* rather than real underlying *needs* when asked “what do you want?”
- Organisational politics and personal agendas can distort what people say.

- Requirements evolve as the project progresses and understanding deepens.

Key idea

The golden rule: don't ask "What do you want?" — ask "What do you DO?" People describe imagined solutions when asked what they want, but reveal real needs, workflows and frustrations when asked about their actual work.

4.7.2 Stakeholders

Stakeholder

Any person, group or organisation with an interest in the system being developed — as a user, funder, regulator, or someone whose work the system affects.

End users	The people who will use the system daily — their needs must drive the requirements
Managers and clients	Fund the project and define business goals; often set priority and scope
Domain experts	Know the business or industry deeply — hospital staff, accountants, legal experts
Technical staff	Developers, DBAs, architects — understand constraints and feasibility
Regulators	Government agencies and standards bodies — set the rules the system must comply with
Operations and support	Maintain the system after delivery — their needs shape maintainability requirements

4.7.3 Fact-finding and domain analysis

Fact-finding	Domain analysis
<i>Elicitation without fact-finding is exploring without a map.</i>	Study the problem domain — its concepts, rules, processes and vocabulary.
Systematically collect information about the organisation, stakeholders and existing systems	Gain deep knowledge of how the business actually works
Identify stakeholders across all levels; interview analysts with domain expertise	Identify domain-specific constraints (laws, standards, best practice)
Study current workflows, tasks and procedures; analyse the existing system if there is one	Discover reusable knowledge from similar systems or previous projects
Assess constraints — cost, technical, policy. Conduct site visits, document reviews, observation	Bridge the knowledge gap between stakeholders and engineers
Outputs: problem context statement, defined system scope (boundaries and interfaces), business objectives, overview of existing processes and stakeholders, preliminary constraints and risks	Benefits: speeds up elicitation, anticipates future needs and changes, reduces miscommunication, encourages reuse of domain models and components

Key idea

If everyone involved agrees up front on the scope of the system, the instability of requirements can be minimised. In continuous-delivery contexts, scope itself is revisited each cycle — but it is still agreed each cycle.

4.7.4 Defining the problem and the scope

Writing a good problem statement	Defining scope
State the <i>problem</i> , not a solution	List everything you might imagine the system doing
Be specific — avoid vague language	Explicitly <i>exclude</i> what is out of scope (this is the hard part)
Name key stakeholders and their pain points	Narrow it if too broad; clarify high-level goals if too narrow
Include measurable impact or expected benefit	Define boundaries and external interfaces
Align with business goals or customer objectives	Agreeing scope up front minimises requirement instability
Keep it short and focused on the core issue	In agile, scope is a negotiable, evolving boundary

Scope questions reshape a project

For a university registration system: is course-catalogue management in scope? Is fee payment? Is timetabling? Each answer changes the size, the team, the architecture and the schedule. A problem can be expressed either as a *difficulty* users face in achieving their goals, or as an *opportunity* to enhance current operations — and the two framings lead to different scopes.

4.7.5 Four elicitation activities

1	Discovery	Engaging with stakeholders to identify needs. Asking questions, listening carefully, observing actual work practices. Most of the learning happens here
2	Classification and organisation	Grouping related requirements. Organising them into functional, non-functional and domain categories
3	Prioritisation and negotiation	Not all requirements can be built, especially not at once. Ranking by importance and resolving conflicts between competing stakeholder needs
4	Specification	Documenting elicited requirements in clear, structured form — user stories, SRS sections or use cases that can be reviewed, tested and traced

4.7.6 Elicitation techniques

Technique	Best for	How it works
Interviews	Rich, qualitative insight from individual stakeholders	One-to-one conversation with open questions. Ask “why” repeatedly to uncover the real need beneath the stated request
Workshops	Bringing multiple stakeholders to agreement	Facilitated group session. Useful for resolving conflicts between stakeholder groups and reaching consensus on scope and priority quickly
Observation	Understanding what people actually do, rather than what they say they do	Watch users work in their real environment. Reveals tacit knowledge that people cannot articulate
Prototyping	Vague requirements — show something to get feedback	Build a quick mock-up. Stakeholders react to what they see, revealing requirements impossible to state verbally (see §2.7)
Document analysis	Regulations, existing systems and domain rules	Review current system manuals, regulations, reports and contracts. Essential for identifying domain requirements
Surveys and questionnaires	Gathering input from many stakeholders quickly	Structured questions distributed widely. Good for quantitative data or reaching stakeholders who cannot attend workshops

4.7.7 The discovery toolkit, by category

Category	Techniques	Best for
Conversational	Interviews, workshops, focus groups, brainstorming	Rich, qualitative understanding; exploring the unknown
Observational	Direct observation, shadowing, ethnography, “day in the life”	Uncovering tacit knowledge — what people actually do versus what they say
Analytical	Document analysis, system archaeology, competitor analysis	Existing systems, regulations, prior art
Synthetic	Prototypes, scenarios, personas, questionnaires and surveys	Validating ideas; reaching many stakeholders at scale

4.7.8 Common challenges

Challenge	How to handle it
Stakeholders cannot articulate their needs	People know what frustrates them but struggle to describe what they want. Show examples, use prototypes, ask “walk me through a typical day”
Different stakeholders want different things	Marketing wants feature X; finance opposes it on cost; legal says it may breach regulation. Conflict is normal — it must be managed, not avoided
Requirements keep changing	New stakeholders join, the business environment shifts, technology evolves. This is why RE is iterative, not a single event at project start
Stakeholders describe solutions, not needs	“I want a button that does X” is a design decision. Ask “why?” until the underlying need is uncovered
Implicit requirements	Things everyone assumes are obvious but nobody wrote down. Example: users assume the system will work on mobile, but nobody said so. Often discovered late

4.8 Requirements analysis and modelling

The purpose of analysis is to refine, organise and analyse elicited requirements so that they are clear, consistent, complete and relevant *before* formal specification.

What analysis achieves	Modelling techniques
Detect inconsistencies, omissions, ambiguities and conflicts	Use case diagrams — functional context and actors
Validate requirements against goals and constraints	Activity diagrams — workflows and process logic
Establish clear relationships among requirements	Class / ER diagrams — structure and data
Reveal gaps that elicitation missed	Data flow diagrams — how data moves and is transformed
Provide the foundation for design, specification and validation	System context diagrams — boundaries and external interfaces
Turn a pile of statements into a coherent system view	State machines — behaviour over time for reactive systems

Key idea

Models are communication tools, not bureaucracy. A single diagram can surface a contradiction that ten pages of prose would hide. All of these notations are covered in Chapter 7.

4.8.1 Requirements classification

Functional requirements	Describe system behaviour and functionality
Non-functional requirements	Define system quality attributes — performance, security, usability
Design constraints	Technology, standards or regulatory limits imposed on the solution
Environmental constraints	Physical, hardware or network conditions the system must operate within

4.9 Prioritising requirements

Budget, time and team capacity are always limited. Prioritisation forces a clear decision about what must be built, what is desirable, and what can wait — and thereby prevents scope creep.

Prioritisation helps you	Criteria to weigh
Decide what to implement first	Benefit to the user — does it solve a real pain?
Distinguish critical from optional features	Cost and effort to develop
Manage stakeholder expectations honestly	Risk and uncertainty
Plan release cycles around value	Dependencies — can it stand alone?
Reduce risk by sequencing the riskiest items early	Ease of deployment and adoption
Avoid treating everything as “high priority”	Strategic or regulatory importance

4.9.1 MoSCoW

M	Must have	Essential. Without this the system will fail or be unusable. Non-negotiable	A hospital booking system <i>must</i> allow patients to book appointments
S	Should have	Important but not vital for the first release. Can be temporarily worked around	It <i>should</i> send confirmation emails — useful, but the system works without it
C	Could have	Desirable. Adds value but is not critical to core function	It <i>could</i> show appointment history. Useful, but patients can manage without it
W	Won't have	Explicitly agreed <i>not</i> to build in this release. Records the decision — not a permanent rejection	It <i>won't</i> integrate with health wearables — deferred to a future version

Exam pointer

The “W” is the most misunderstood letter. It does not mean “rejected”; it means “consciously deferred, and everyone has agreed”. Recording the decision is what stops the item being re-argued every sprint.

4.9.2 Other prioritisation techniques

Kano model	Classifies features by their effect on satisfaction: <i>basic</i> (expected; absence causes dissatisfaction), <i>performance</i> (more is better, linearly), and <i>delighters</i> (unexpected features that create disproportionate satisfaction). Useful for arguing that a basic feature outranks an exciting one
Weighted Shortest Job First	Rank by <i>cost of delay</i> divided by <i>job size</i> . Used in SAFe. Favours small items with high urgency, which maximises value delivered per unit time
Ranking / 100-point method	Stakeholders distribute a fixed budget of points across requirements, forcing genuine trade-offs rather than universal high priority

Key idea

The hardest truth in prioritisation: if everything is a priority, nothing is. Forcing a ranking *is* the value — it makes trade-offs explicit and accountable.

4.10 Requirements specification

Requirements specification

The activity of documenting the discovered and analysed requirements in a clear, structured form that all stakeholders can understand, and that developers, testers and designers can work from.

Level of formality	Character	Example
Natural language	Plain sentences — accessible to everyone but can be ambiguous. Most common for user-level requirements	“The system shall allow registered users to reset their password via email.”
Structured language	Templates and fixed formats that reduce ambiguity while staying readable. Good for system requirements	ID: UC-004 · Actor: Registered user · Trigger: user clicks ‘Forgot password’ · Post-condition: reset email sent
Graphical models	Diagrams showing structure and flow visually. Excellent for communication	A use case diagram showing ‘Reset Password’ with the Registered User actor and the Email System actor

4.10.1 Writing one requirement well

User need (rough)	“As a patient, I want to reset my forgotten password so that I can log in again.” <i>Clear intent — but who can do it? How? What limits? All unstated.</i>
System requirement	“The system shall allow a registered patient to request a password reset and shall email a reset link to the patient’s registered address.” <i>Now precise about who, what and how. But missing measurable limits.</i>
Refined and testable	“... The reset link shall expire 24 hours after issue and shall be usable only once.” <i>Now testable. A developer knows exactly what to build; a tester can verify it.</i>

4.10.2 The Software Requirements Specification

SRS

A complete description of the software system to be built — covering what it must do (functional), how well it must do it (non-functional), and the constraints it must operate under. It is the primary reference document for the whole project.

A good SRS is **clear** (one valid interpretation of every requirement), **complete** (all functional, non-functional and domain requirements are present), **consistent** (no requirement contradicts another; terms are used identically throughout), **verifiable** (every requirement can be tested) and **traceable** (each requirement links back to a stakeholder need and forward to a design element or test case).

Typical SRS structure (based on the IEEE standard)

1	Introduction	Purpose and scope of the system, intended readers, definitions and abbreviations, references, document overview
2	Overall description	System context, the product's place in the wider environment, user characteristics, assumptions and constraints
3	User requirements	What users need, in plain language or as use cases. Readable by non-technical stakeholders including the client
4	System requirements	Detailed functional and non-functional requirements, written precisely for developers, architects and testers
5	Models and diagrams	Use case diagrams, data flow diagrams, entity-relationship diagrams and other visual models
6	Appendices	Glossary, references, version history, supporting information not in the main body

When to use a full SRS

Safety-critical systems · regulated industries · fixed-price contracts · large distributed teams who need a shared written reference.

4.10.3 The product backlog — agile specification

In agile, the backlog *is* the living requirements document. It is an ordered, ever-evolving list of everything the product might need. It is never “finished”: top items are detailed and ready to build; items further down are deliberately rough.

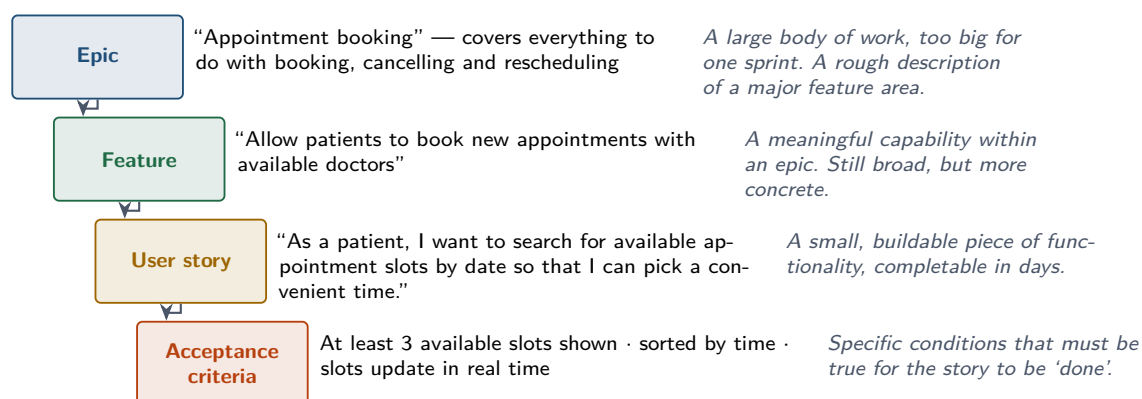


Figure 4.4: The agile requirements hierarchy: detail increases as an item approaches implementation.

4.10.4 SRS versus product backlog

	SRS (plan-driven)	Product backlog (agile)
When written	All upfront, before development	Incrementally, just before needed
Who reads it	All stakeholders, especially developers and testers	Development team and Product Owner
How detailed	Complete and precise — a formal document	Top items detailed; lower items deliberately rough
How it changes	Formal change control with reviewed approval	Continuous reprioritisation — flexible
Document form	Structured chapters, numbered requirements	Ordered list of stories with acceptance criteria
Best for	Regulated, contractual, safety-critical work	Evolving requirements, fast feedback loops

Key idea

Neither approach is universally better. Many organisations use both on different parts of the same project — an SRS for the regulated core, a backlog for the customer-facing layer.

4.10.5 Common problems in requirements documents

Problem	Wrong	Right
Mixing design and requirements — writing <i>how</i> it will be built instead of <i>what</i> it must do	✗ “The system shall store passwords using bcrypt hashing.”	✓ “The system shall protect user passwords against unauthorised access.”
Vague, untestable language — “fast”, “user-friendly”, “robust”, “flexible”	✗ “The system shall be user-friendly.”	✓ “A new user shall complete registration in under 3 minutes.”
Missing requirements — something “everyone knows” was never written down	✗ Deployment of an app with no mention of mobile browser support	✓ “The system shall function correctly on Chrome, Safari and Firefox on mobile devices.”
Inconsistency — one section contradicts another	✗ §3: “All users must log in.” §5: “Guest users may access the catalogue.”	✓ One term, one definition, used consistently throughout

4.11 Requirements validation

Validation asks two questions of the requirements themselves: are these the *right* requirements, and are they *good enough* to build from? (The general distinction between verification and validation is set out in §2.6.)

4.11.1 The 5C check

Correct	Do the requirements reflect the stakeholders’ real needs — not misunderstandings?
Complete	Are all necessary requirements present? Have any important ones been left out?
Consistent	Are there any contradictions between requirements?
Capable	Are the requirements technically and financially feasible to implement?
Checkable	Can each requirement be verified or tested? If not, it must be rewritten.

4.11.2 Validation techniques

Requirements reviews	A structured meeting where stakeholders read through requirements together, looking for errors, inconsistencies and gaps. Can be formal (with chair and recorder) or informal
Prototyping	Build a quick, rough version of the interface and let real users react. Reveals requirements that cannot be articulated in words but become obvious when seen
Test-case generation	For each requirement, try to write a test case. If you cannot write a clear test, the requirement is ambiguous or incomplete and must be rewritten before acceptance
Acceptance criteria	In agile contexts, requirements are validated through concrete acceptance criteria written <i>before</i> development begins. If the feature meets all criteria, the requirement is satisfied
Walkthroughs	The author presents requirements step by step to a small group who ask questions and identify issues. Less formal than a review; good for early-stage requirements

4.12 Requirements management

4.12.1 Why requirements change

Evolving business needs	Business goals, regulations and market conditions change over time — so must the system that supports them
Stakeholder diversity	Different users and funders discover they want different things as the system takes shape
Technology shifts	New platforms, APIs or integration requirements emerge that were not anticipated at planning
Understanding deepens	What the team needed at the start often differs from what they discover after working with the system

Key idea

“Walking on water and developing software from a frozen specification are equally easy — if both are frozen.” (Ed Yourdon). Change is not failure; it is the normal condition of software projects. A good process *manages* change rather than pretending it will not happen.

4.12.2 Two pillars

Pillar 1 · Change management	Pillar 2 · Traceability
Establish a baseline — an agreed, approved set of requirements	Every requirement is linked to the stakeholder need that drove it
When change is proposed, evaluate impact, cost and risk before deciding	Every requirement links forward to a design decision and a test case
Only implement approved changes — no informal, uncontrolled changes	Enables impact analysis: “if this changes, what else breaks?”
In agile, the backlog <i>is</i> the living baseline; reprioritisation is continuous	Helps detect missing features, and features with no requirement (scope creep)

4.12.3 The change management process

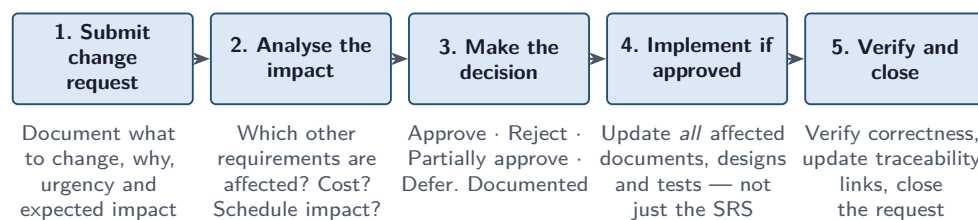


Figure 4.5: The change management process. Step 4 is where projects most often fail — updating the SRS but not the tests, the design or the traceability matrix.

4.12.4 Traceability

Traceability

The ability to follow a requirement’s life in both directions — from the original stakeholder need that created it, through to every design decision, code component and test case that implements and verifies it.

Traceability matters for three concrete reasons: impact analysis tells you exactly what else must change when a requirement changes; features with no linked requirement are unauthorised additions (scope creep); and auditors in regulated industries can verify that every legal requirement is implemented and tested.

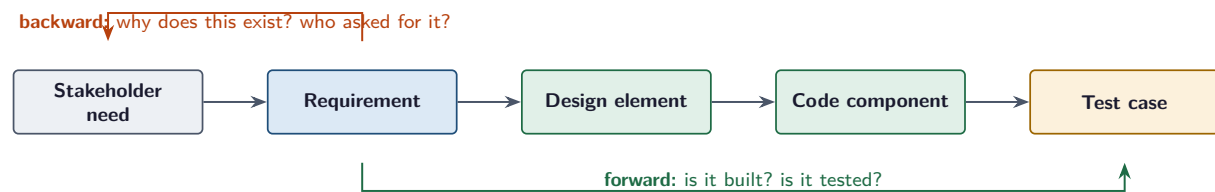


Figure 4.6: Backward and forward traceability.

4.13 AI in requirements engineering

Requirements engineering is largely *language* work — reading interviews, writing documents, detecting ambiguity, reformatting between specifications. These are exactly the tasks modern AI handles well.

AI does well	Humans remain essential
Detecting ambiguous or vague language	Deciding which stakeholder's need takes priority
Drafting user stories from rough notes	Negotiating conflicts between stakeholders
Suggesting acceptance criteria for a story	Ethical and legal judgement on what is acceptable
Summarising long interview transcripts	Building trust and reading the room with stakeholders
Checking requirements against a standard format	Accountability — if a requirement fails, the engineer owns it

4.13.1 Cautions

AI can invent requirements	AI may generate plausible-sounding requirements no stakeholder ever expressed. Always trace every requirement back to a real person or a real source document
Fluent output feels authoritative	Well-written AI output looks convincing. Teams sometimes accept it without the scrutiny they would apply to a colleague's work
AI cannot validate its own output	If AI both writes a requirement and checks it, it simply confirms its own assumptions. Always have an independent human review
Confidentiality risk	Requirements often contain sensitive business and personal information. Check the tool's data-handling policy before pasting anything into a public AI service
The engineer is still responsible	If an AI-suggested requirement causes the system to fail, the engineer who accepted it is accountable

4.14 Common pitfalls in requirements engineering

Solutioning too early	Capturing a stakeholder's proposed solution instead of their underlying need. Always ask "why?" to get back to the real problem
Ambiguity left unresolved	Vague terms mean different things to different people — discovered only at acceptance time, when it is expensive
Missing non-functional requirements	Specifying what the system does but not how well. Then discovering at launch that it is too slow, insecure or unusable under real load
Forgetting some stakeholders	Building what one group wants while ignoring others — discovered when the delivered system is rejected by users
The document becomes the goal	Optimising for a complete, polished SRS rather than genuine shared understanding. The artefact is a means, not the end
No change management	No baseline, no traceability, no impact analysis — every change becomes chaos and the project loses control of its own scope

Software Design

Learning outcomes for this chapter

- LO1.** Explain what software design is and its role in the lifecycle.
- LO2.** Describe the design journey from high-level architecture down to detailed component design.
- LO3.** Explain the core design principles — abstraction, modularity, cohesion, coupling and information hiding.
- LO4.** Describe the major design approaches and when each applies.
- LO5.** Describe how design differs in plan-driven and agile contexts.
- LO6.** Explain what a design specification records, and identify tools and AI assistance available.

The running example — the UCSC Student Portal

Chapters 5, 6 and 7 use one system throughout, so that each concept is illustrated against something already familiar. The portal has five features:

Course registration	Students register for courses each semester within their degree programme
Results and grades	Students view marks; lecturers upload results; the registry verifies them
Fee payment	Students pay tuition and library fees; the finance office manages accounts
Timetable	Students and lecturers view weekly class and examination schedules
Facility booking	Students book laboratories, sports facilities and meeting rooms online

5.1 What is software design?

Software design

The activity that transforms *what* the system must do (requirements) into *how* it will be built — a blueprint defining the parts of the system, how they connect, and what each part is responsible for, before any code is written.

Design and requirements inform each other. Designing a system reveals requirements that need clarification. In agile they happen concurrently inside each sprint.

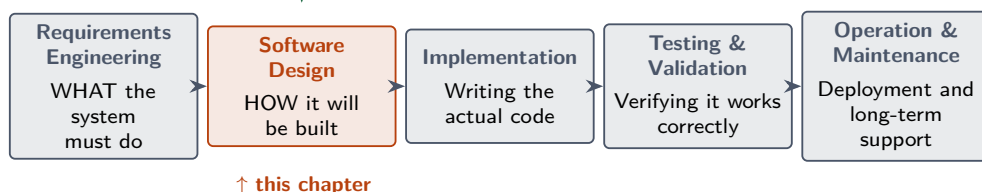


Figure 5.1: Design in the software lifecycle.

5.1.1 What makes a design good?

Correct	Satisfies all requirements — functional and non-functional
Simple	No unnecessary complexity — the simplest design that meets the requirements
Modular	Broken into independent pieces; you can change one without breaking others
Cohesive	Each part does one thing well, not a random mix of unrelated logic
Loosely coupled	Parts depend on each other as little as possible
Documented	Design decisions are recorded so others can understand and maintain the system

5.2 The design journey — top down

In top-down design you start at the highest level — the whole system — and progressively refine it until each part is small enough to code directly.

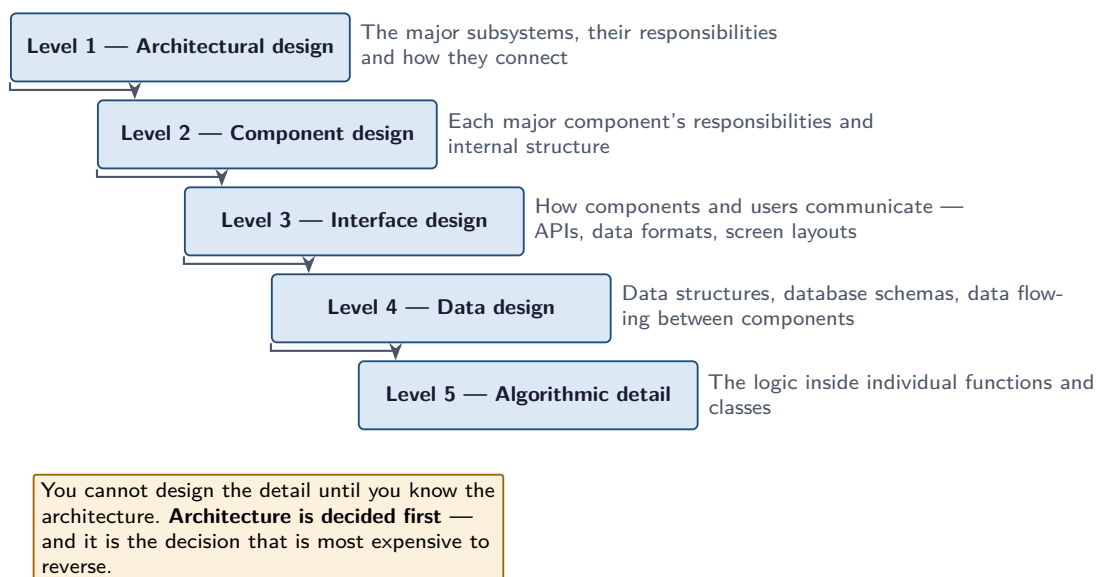


Figure 5.2: Five levels of design refinement.

5.2.1 Level 1 — architectural design

From the requirements we know the portal needs five features. The first design decision is how to organise them into major subsystems.

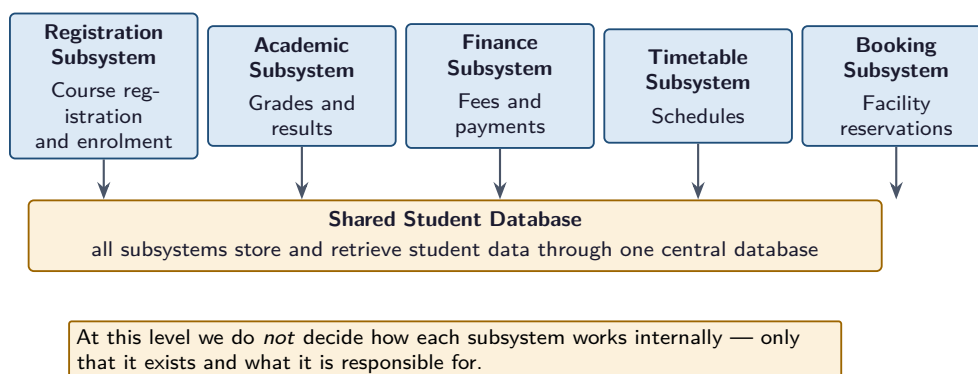


Figure 5.3: Level 1: the Student Portal's five subsystems.

5.2.2 Level 2 — component design

Zooming into one subsystem: what components does the Registration Subsystem contain, and what is each responsible for?

Component	Responsibility
RegistrationController	Receives student requests, validates them, and coordinates the overall flow
CourseAvailabilityChecker	Checks whether a course has open places and whether prerequisites are met
TimetableConflictChecker	Checks whether the selected course clashes with already-registered courses
EnrolmentManager	Records the enrolment and triggers the confirmation notification
RegistrationDatabase	Stores all enrolment records persistently

5.2.3 Level 3 — interface design

Interface design is not only API design. It has two distinct sides, and confusing screens with APIs is a common student error.

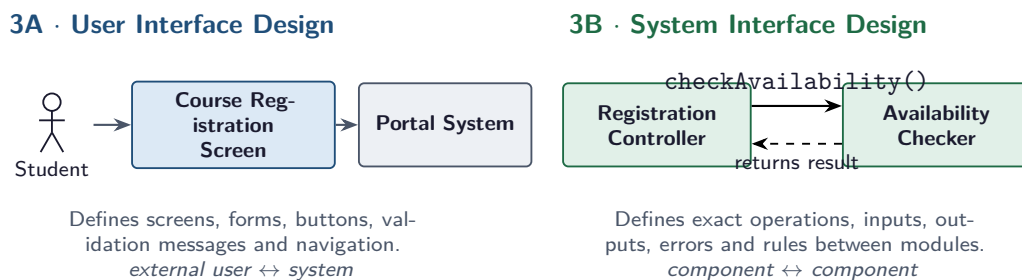


Figure 5.4: Level 3 has two audiences: people and software.

	3A · User interface	3B · System interface
Focus	How external users interact through screens, forms, navigation and feedback	How modules, services and databases communicate through contracts and messages
Typical evidence	Wireframes, screen catalogue, field rules, messages, accessibility and responsive behaviour	API signatures, request/response formats, preconditions, postconditions, errors, security rules
Portal example	Course Registration screen: search course, select semester, register, show success or error	<code>checkAvailability(studentID, courseCode) → { available, reason }</code>

A system interface contract in full

Operation: `checkAvailability(studentID: String, courseCode: String)`

Returns: `{ available: Boolean, reason: String }`

Provided by: `CourseAvailabilityChecker` **Consumed by:** `RegistrationController`

Request example: `studentID = S1001, courseCode = CS1101`

Response example: `available = true`

Error contract — possible failures: course does not exist · course is full · prerequisite not met · timetable conflict.

Every caller now knows exactly what to send, what to expect back, and what can go wrong — *without knowing anything about the internals.*

5.2.4 Level 4 — data design

Data design defines how business information is organised, protected and changed. Poor data design can break registration, reporting, security and future integration simultaneously.

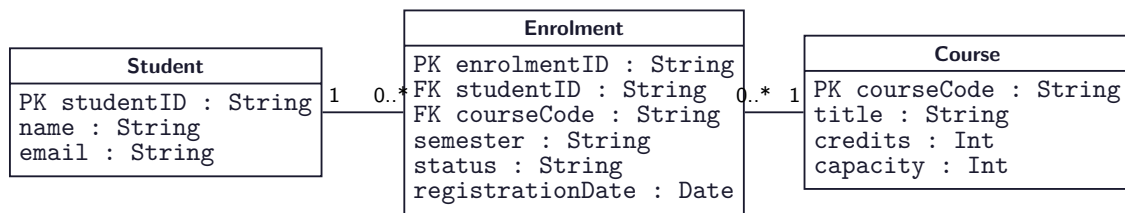


Figure 5.5: Level 4: the logical data model for enrolment.

Table: Enrolment

Column	Type	Rule
enrolmentID	String	Primary key
studentID	String	Foreign key → Student
courseCode	String	Foreign key → Course
semester	String	Required
status	String	Pending / Confirmed / Cancelled
registrationDate	Date	Required

Data design also records constraints, indexes, transactions, ownership, security, backup and data-retention rules. Two examples specific to the portal:

- **Design rule.** A student can have many enrolments. One enrolment must reference exactly one student and exactly one course.
- **Transaction rule.** When a student registers, the availability check and the enrolment insert must complete safely as one controlled (atomic) operation — otherwise two students can be granted the last remaining place.

5.3 Core design principles

A warning story

A student writes all five portal features in one 3,000-line file. It works at first. Six months later nobody understands it, a small change to fee logic breaks course registration, and three developers cannot work on it simultaneously. Design principles exist to prevent exactly this.

Abstraction	Focus on the essential; hide unnecessary detail
Modularity	Divide the system into independent, manageable pieces
Cohesion	Each module does <i>one</i> thing — not many unrelated things
Low coupling	Minimise dependencies between modules
Information hiding	Hide internal implementation; expose only what is needed

5.3.1 Abstraction

Abstraction

Representing a system by focusing only on what is relevant for a particular purpose, suppressing everything else. You work with a simplified model of reality.

User level	“I click Register for CS1101” — the student has no idea what happens underneath
Interface level	<code>registerForCourse(studentID, courseCode)</code> — the controller calls this without knowing how it works
Logic level	Check availability → check timetable → check fees → add record → send email
Data level	<code>INSERT INTO Enrolments VALUES (studentID, courseCode, date, status)</code>

Abstraction lets each developer work at their own level without being overwhelmed by the whole system at once.

5.3.2 Modularity

Without modularity

One massive portal file — all five features together
 Change fee logic → risk breaking registration
 Three developers cannot work simultaneously
 Testing one feature requires running everything

With modularity

Five separate subsystems, each independent
 Change finance without touching registration
 Three teams build three modules in parallel
 Test each module in isolation — fast and safe

Key idea

A modular system is like Lego bricks — you can add, remove or replace pieces without rebuilding everything else.

5.3.3 Cohesion

Cohesion

A measure of how closely related the responsibilities within a single module are. A highly cohesive module has one clear purpose; every piece of code inside contributes to that single purpose. **High cohesion is good.**

High cohesion — `TimetableConflictChecker`

```
checkForConflict()
getStudentSchedule()
compareTimeslots()
returnConflictDetails()
```

All four functions serve *one* purpose: checking timetable conflicts.

Low cohesion — `StudentModule`

```
checkTimetableConflict()
processFeePayment()
sendResultsEmail()
bookSportsFacility()
```

Four completely unrelated responsibilities in one module. A change to fees might affect timetables.

5.3.4 Coupling

Coupling

A measure of the degree of interdependence between modules. Low (loose) coupling means a change in one module is unlikely to force a change in another. **Low coupling is good.** High (tight) coupling creates fragile systems where changes cascade everywhere.

Tight coupling

`RegistrationController` reads Finance database tables directly
 If the Finance schema changes, Registration code breaks too
 Cannot test Registration without the Finance system running
 Developers from both teams must coordinate every change

Loose coupling

`RegistrationController` calls `checkFeeStatus(studentID)` only
 The Finance team can change internals freely without affecting Registration
 Teams develop and test independently
 Replace the Finance module entirely — Registration never knows

Exam pointer

These two are examined together and are easy to confuse. A memory hook: *cohesion is about the inside of a module; coupling is about what happens between modules*. Both improve when a module has one clear purpose and communicates only through a narrow, well-defined



Figure 5.6: The two scales point in opposite directions. Aim for *high* cohesion within a module and *low* coupling between modules.

interface.

5.3.5 Information hiding

Information hiding

The internal implementation details of a module are hidden from the outside world. Other modules see only the interface — the inputs and outputs — not how the work is done.

Freedom to change	Rewrite <code>EnrolmentManager</code> to use a different database; as long as the interface stays the same, nothing else changes
Simpler to use	Developers using <code>EnrolmentManager</code> only need to understand <code>saveEnrolment(studentID, courseCode)</code> , not 300 lines of internal SQL
Prevents misuse	If internal data is not exposed, external code cannot accidentally corrupt it. The database connection is private

5.3.6 Separation of concerns — the shared goal

All five principles serve one goal: every part of the system deals with its own concern and is insulated from changes in other concerns.

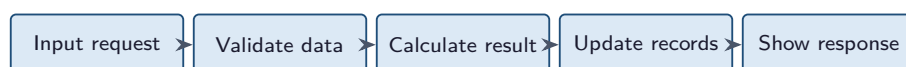
Fee processing concern	<i>Only</i> the Finance Subsystem deals with fee logic. If the government changes the tax rate, only Finance changes. Registration, Grades and Booking are unaffected
Timetable concern	<i>Only</i> the Timetable Subsystem manages schedule data. When a new semester is added, only Timetable is updated
Data storage concern	<i>Only</i> the data layer knows <i>how</i> data is stored. Switch database vendor and nothing else changes
User interface concern	<i>Only</i> the presentation layer deals with screens. Redesign the portal's look completely — the business logic is untouched

5.4 Major design approaches

A design approach is a way of *thinking* about how to organise a system. There is no single best approach — real projects combine several, choosing the lens that answers the current design question.

5.4.1 1. Procedural / workflow design

Design the system as a sequence of procedures or steps.



Main question	What steps must happen, and in what order?
Good for	Simple workflows, algorithms, scripts, batch processing, step-by-step business processes
Limitation	As the system grows, many procedures share global data and it becomes hard to maintain
Portal example	“Pay fees” designed first as a procedure: log in → view fees → select method → pay → record payment → show receipt
Related diagram	Activity diagram, sequence diagram (§7.6, §7.5)

5.4.2 2. Object-oriented design

Design the system around objects that combine data and behaviour.

Student	Course	Enrolment	Payment
- studentID - name - email	- courseCode - title - credits	- enrolmentID - date - status	- paymentID - amount - status
+ register() + viewResults()	+ getAvailability()	+ confirm() + cancel()	+ process() + refund()

Main question	What are the important things in the system, what data do they hold, and what behaviour belongs to each?
Why useful	Encapsulation keeps related data and operations together, improving maintainability and reuse
Watch out	Do not create classes merely because nouns appear in the requirements. Every class must have a clear responsibility
Related diagram	Class diagram (§7.4)

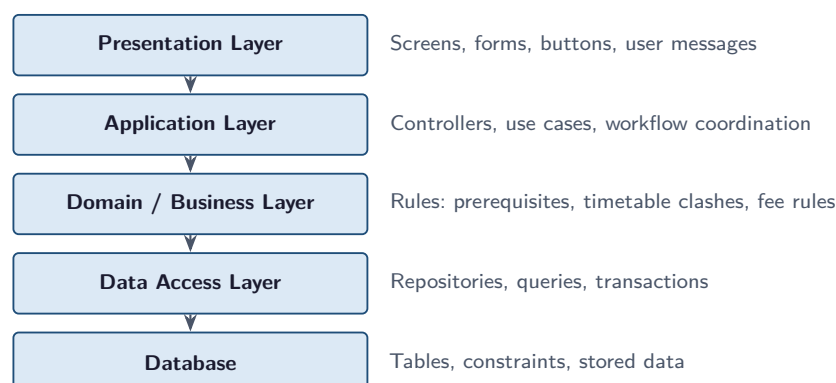
5.4.3 3. Component-based design

Design the system as independent modules with clear interfaces.

Main question	How can the system be divided into replaceable, testable, independently understandable modules?
Key design output	Component responsibilities, required and provided interfaces, dependencies, and ownership
Portal example	Registration uses CourseAvailabilityChecker and EnrolmentManager through defined interfaces
Related diagram	Component diagram; HLD/LLD documents

5.4.4 4. Layered / tiered design

Separate responsibilities so that each layer has a clear job and dependencies point in one direction only.



Key idea

UI changes should not require database redesign, and database changes should not directly affect every screen. Layers exist to control the *direction* of dependency.

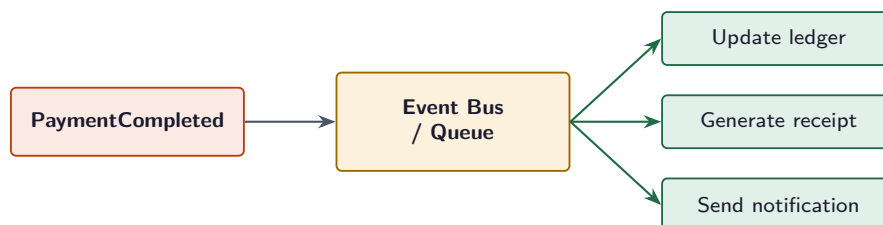
5.4.5 5. Service-oriented / API-based design

Design separate services that communicate through defined APIs.

When useful	Different teams own different areas; services must be changed and deployed separately
Portal example	POST /registrations · GET /courses/{courseCode}/availability · POST /payments · POST /notifications
Important warning	More services add communication, testing, monitoring and deployment complexity
Microservices	A microservice is a smaller, <i>independently deployable</i> service — not just a small class

5.4.6 6. Event-driven design

Components communicate by publishing and reacting to events.



Main question	What important events happen in the system, and which components should react?
Good for	Notifications, audit logs, background processing, integration with external systems
Watch out	Event flows become hard to trace unless events, ownership and failure handling are documented

5.4.7 7. Functional and data-centred design

Functional design idea	Data-centred design idea
Design logic as small, predictable functions	Design around the data model, database rules and data ownership
<code>validatePrerequisites(student, course) → true/false</code>	Entities: Student, Course, Enrolment, Payment, Result.
<code>calculateGPA(results) → GPA</code>	Questions: who owns this data? what constraints protect it? what transaction must be atomic?
<code>calculateOutstandingFees(student) → amount</code>	

Functional design is rarely the whole architecture of a system like the Student Portal, but it is very useful for validation rules, calculations and transformations.

5.4.8 Choosing an approach

If the design question is...	Use this approach	Useful diagram / document
What steps happen?	Procedural / workflow	Activity diagram, sequence diagram
What things and relationships exist?	Object-oriented	Class diagram
How do we split the system?	Component-based	Component diagram, HLD/LLD
Where should responsibilities live?	Layered / tiered	Architecture diagram
How do modules communicate?	API / system interface	Interface specification
How is persistent data organised?	Data-centred	ERD, database design document

Key idea

Choose the design lens that answers the current design question — not the one that is fashionable. The Student Portal ends up using use cases for user goals, classes for data and behaviour, components for subsystems, interfaces for communication, and database design for persistence, all at once.

5.5 Design in both cultures

Key idea

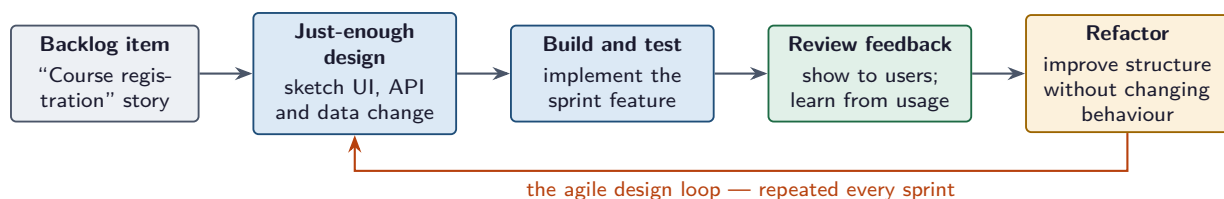
Agile does not remove design. It changes how design is discovered, documented, reviewed and improved. Compare the cultures by *timing*, *documentation depth*, *feedback* and *change control* — never as “design” versus “no design”.

5.5.1 Plan-driven design — complete blueprint before build

1	Requirements specification	Complete SRS signed off by stakeholders
2	High-Level Design (HLD)	Architecture, major subsystems, patterns, deployment decisions
3	Detailed Design (LLD)	Component internals, interfaces, UI screens, data schema
4	Design review and sign-off	Stakeholders approve the design before coding starts
5	Implementation	Developers build according to the approved design

In plan-driven development, design is treated as a formally approved *contract* before implementation begins. It is useful when requirements are stable, risk is high, or formal approval is required.

5.5.2 Agile design — enough design, then continuous improvement



Architecture remains important; the detail emerges sprint by sprint. Three mechanisms protect the architecture while keeping the design light:

Architecture spike		A short, timeboxed investigation or prototype used to answer a key technical question before committing
Architecture runway		A minimal agreed architecture that supports the upcoming sprints without over-designing everything
Architecture Record	Decision	A short note recording context, decision and consequences. Lightweight but traceable

5.5.3 Side by side

Aspect	Plan-driven design	Agile design
Timing	Most design completed before coding	Design done just in time for the current sprint
Level of detail	Full HLD, LLD, UI, API and database detail early	High-level direction early; details emerge incrementally
Documentation	Formal SDD/HLD/LLD documents and approval	Lightweight evidence: sketches, ADRs, tests, key diagrams
Change control	Formal change request; changes can be costly	Change welcomed; design adapts through refactoring
Risk handling	Predict risks early and plan mitigation	Discover risks early using spikes and short feedback loops

Same feature, two design cultures — course registration

Plan-driven delivery

- 1. Before coding** — complete UI screens, API contracts, database schema and sequence diagrams
- 2. Approval** — review and sign off the SDD before implementation
- 3. Coding** — build exactly according to the approved design

Agile delivery

- Sprint 1** — search and view available courses; lightweight UI and API design
- Sprint 2** — add register action; refine the Enrolment table and validation rules
- Sprint 3** — add conflict checking, error messages, logging and refactoring

Both cultures design the same things — architecture, UI, interfaces and data — with different timing and documentation weight. Design is a *decision-making* activity; the process culture decides when, how much and how formally those decisions are recorded.

5.6 Design specification

Design specification

The record of design decisions, so that developers, testers and maintainers can understand and implement the system correctly. It is the bridge between thinking about the design and writing the code.

Document		Contents	Audience
High-Level (HLD)	Design	The architectural view — subsystems, patterns and how they connect	Architects, senior developers, project managers
Low-Level (LLD)	Design	Component internals — class structures, algorithms, interface contracts	Developers who will write the code
Software Design Document (SDD)		The complete specification: HLD + LLD + UML diagrams + design rationale	The whole project team

5.6.1 Design specification in agile

User stories with acceptance criteria	The requirement and design intent in one place; acceptance criteria define testable behaviour
Quick whiteboard diagrams	A rough class or sequence diagram from a planning session, photographed and stored — it does not need to be perfectly formatted
Architecture Decision Records	Short records of major decisions: what was decided, why, what alternatives were considered. Preserves reasoning for future teams
Definition of Done	Encodes the quality bar that all design work must meet before it counts as complete

Key idea

“Documentation is not the goal — shared understanding is. Use the minimum documentation needed to achieve it.”

5.7 Tools and AI in software design

Diagramming tools	draw.io (free), Lucidchart, Microsoft Visio — UML diagrams, architecture sketches, data flow diagrams
IDEs with AI assistance	Visual Studio Code, IntelliJ IDEA — with GitHub Copilot and similar completion built in
Architecture and modelling	Enterprise Architect, StarUML — large-scale UML modelling and architecture documentation
Collaborative design	Miro, Figma — virtual whiteboards for remote teams designing together in real time
AI design assistants	ChatGPT, Claude, GitHub Copilot — generate design suggestions, review architecture, write class structures, explain patterns

Where AI helps in design today

Code and class generation	Copilot and Cursor suggest class structures, method signatures, boilerplate and patterns as you type
Architecture review	Describe your architecture; AI identifies potential problems, suggests improvements and explains trade-offs
UML diagram generation	Tools like PlantUML with AI turn a text description into a diagram in seconds
Design pattern suggestions	Describe a design problem; AI suggests which pattern fits and shows example code for your context
Refactoring assistance	Paste code and ask AI to identify cohesion and coupling problems and suggest improvements

What AI cannot do

Evaluate business context	AI does not know your users, organisation, constraints or business goals. The engineer must supply that context and judge fit
Resist over-engineering	AI often suggests complex solutions. Simplicity still applies — the engineer judges whether the design is the <i>simplest</i> that meets requirements
Substitute for understanding	Never paste AI-generated design into a system you do not understand. You are responsible for the design
Avoid propagating bad habits	AI is trained on vast amounts of existing code, including poorly designed code, and can confidently suggest designs that violate cohesion, coupling or information hiding
Take accountability	If an AI-suggested architecture leads to failure, the engineer who accepted it is responsible

5.8 Common design pitfalls

Skipping the architecture	Starting to code before deciding the structure. Discovered late, when structural change is expensive and disruptive
Over-engineering	Designing for every possible future scenario. Complexity added now must be maintained forever. Design for what is needed — not for every “what if”
God class / god function	One class or function that does everything. Zero cohesion, maximum coupling. The first sign of design breakdown
Tight coupling everywhere	Every component knows the internals of every other. A change anywhere breaks everything. The second sign of breakdown
Skipping design documentation	“I’ll just write the code” — three months later nobody knows why anything was built the way it was
No design review	Problems that would take ten minutes to fix at the design stage take weeks to fix in code

Software Architecture

Learning outcomes for this chapter

- LO1.** Explain why the architectural design of a system is important.
- LO2.** Describe the decisions that must be made during architectural design.
- LO3.** Describe common architectural patterns and explain what problem each solves.
- LO4.** Explain how application-specific architectures apply to transaction-processing and language-processing systems.

6.1 What is software architecture?

Software architecture (IEEE 1471)

The fundamental organisation of a system — its major components, the relationships between them, and the principles guiding their design and evolution.

Architecture	The big picture: the structure, patterns and constraints of the whole system
Design	Decisions inside that structure: component internals, interfaces, data
Code	The realisation of those designs: the actual lines of software

6.1.1 Why architecture must come first

Architecture determines quality	Performance, security and scalability are determined largely by architecture, not by individual functions
Architecture shapes team structure	Team organisation follows system structure. Define the architecture and you define how teams work
Architecture is expensive to change	Refactoring a class is cheap; restructuring the fundamental organisation of a system can cost months
Architecture enables parallel work	Once component boundaries are defined, many developers can work simultaneously without conflict

6.1.2 Architectural design decisions

Architectural design is a creative process, but a set of common questions recurs in every project:

- Is there a generic application architecture that can act as a template?
- How will the system be distributed across a number of cores or processors?
- What architectural patterns or styles might be used?
- What will be the fundamental approach used to structure the system?
- How will the structural components be decomposed into sub-components?
- What strategy will be used to control the operation of the components?
- How should the architecture be documented and evaluated?

6.1.3 Architecture and non-functional requirements

Performance	Localise critical operations, minimise communication, use large rather than fine-grain components
Security	Use a layered architecture with the most critical assets in the innermost layers
Safety	Localise safety-critical features in a small number of subsystems
Availability	Include redundant components and mechanisms for fault tolerance
Maintainability	Use fine-grain, replaceable components and avoid shared data structures

Exam pointer

Note the direct conflicts. Performance favours *large* components; maintainability favours *fine-grain* components. Security favours layering; performance penalises it. Architectural design is therefore an exercise in *trade-off*, and a good exam answer names the trade-off explicitly rather than listing benefits.

6.1.4 Architectural views

No single diagram can express everything about an architecture. Four complementary views (Kruchten's "4+1" model) are conventional:

Logical view	The key abstractions in the system as objects or object classes; relates requirements to entities
Process view	How the system is composed of interacting processes at run time; useful for performance and availability judgements
Development view	How the software is decomposed for development — the view software engineers actually work in
Physical view	System hardware and how software components are distributed across processors
+1: Use cases	The scenarios that tie the other four views together and validate them

6.2 Architectural patterns

Architectural pattern

A stylised description of good design practice that has been tried and tested in different environments. Patterns are a means of representing, sharing and reusing knowledge. A pattern description should include information about when the pattern is — and is not — useful.

6.2.1 Pattern 1 — Layered architecture

Name	Layered architecture
Description	Organises the system into layers with related functionality associated with each layer. A layer provides services to the layer above it, so the lowest-level layers represent core services likely to be used throughout the system
Example	A layered model of a system for sharing copyright documents held in different libraries
When used	When building new facilities on top of existing systems; when development is spread across several teams with each team responsible for a layer; when there is a requirement for multi-level security
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (for example authentication) can be provided in each layer to increase dependability
Disadvantages	Providing a clean separation between layers is often difficult in practice, and a high-level layer may have to interact directly with lower-level layers. Performance can suffer because of multiple levels of interpretation as a request is processed at each layer

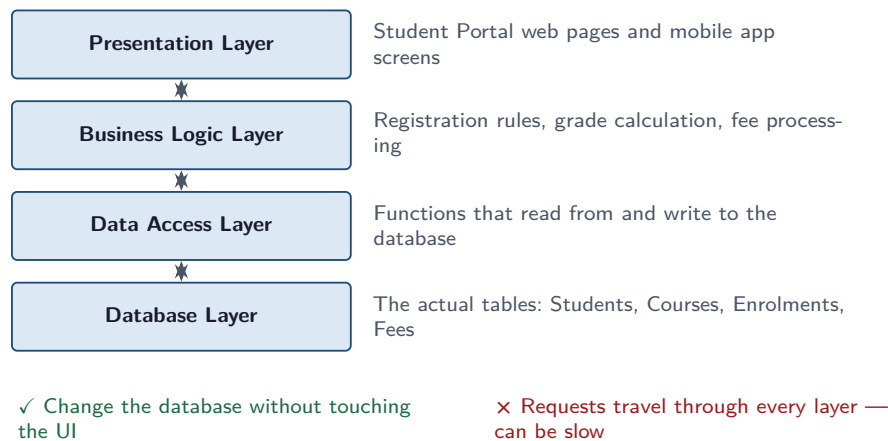


Figure 6.1: Layered architecture applied to the Student Portal. Each layer provides services to the layer above and uses services from the layer below; layers communicate only with adjacent layers.

6.2.2 Pattern 2 — Client-server

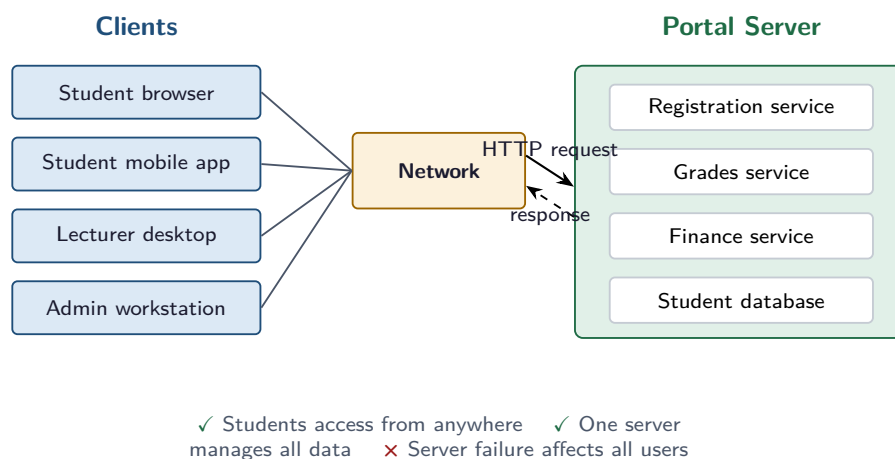
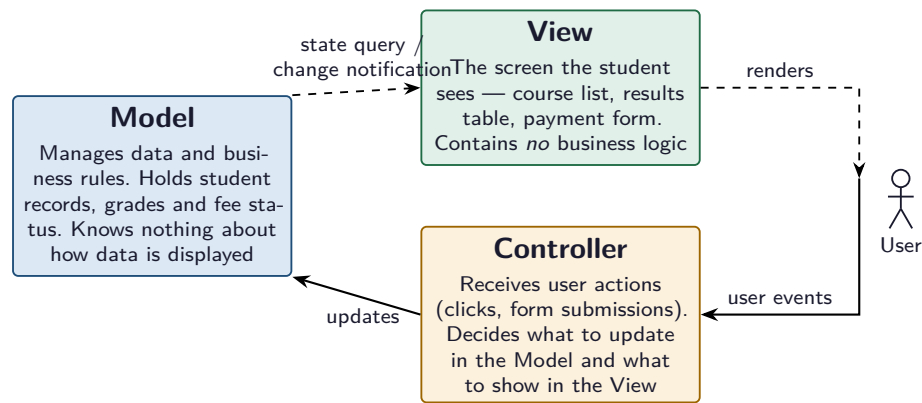


Figure 6.2: Client-server architecture.

Name	Client-server
Description	The functionality of the system is organised into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them
Example	Different users accessing a website on the server using their individual machines
When used	When data in a shared database has to be accessed from a range of locations. Because servers can be replicated, it may also be used when the load on a system is variable
Advantages	Servers can be distributed across a network. General functionality (for example a printing service) is available to all clients and need not be implemented by all services
Disadvantages	Each service is a single point of failure, so it is susceptible to denial-of-service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. There may be management problems if servers are owned by different organisations



✓ Change the UI without touching data logic ✓ The same data displayed many ways Used in Django, Laravel, Spring MVC, Rails

Figure 6.3: Model-View-Controller.

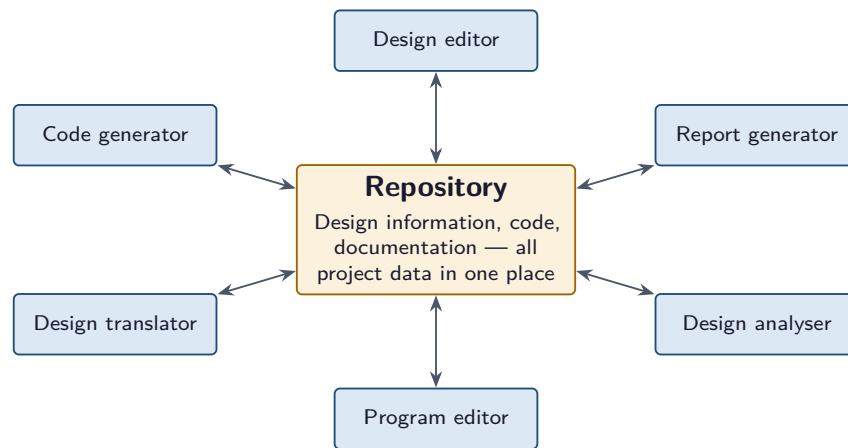
6.2.3 Pattern 3 — Model-View-Controller

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components. The <i>Model</i> manages the system data and associated operations; the <i>View</i> defines and manages how data is presented to the user; the <i>Controller</i> manages user interaction and passes these interactions to the View and the Model
Example	The architecture of a web-based application system organised using the MVC pattern
When used	When there are multiple ways to view and interact with data; also when the future requirements for interaction and presentation of data are unknown
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways, with changes made in one representation shown in all of them
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple

6.2.4 Pattern 4 — Repository

Subsystems must exchange data, and this may be done in two ways: shared data is held in a central repository accessible to all subsystems, or each subsystem maintains its own database and passes data explicitly to others. When large amounts of data are shared, the repository model is the most commonly used and most efficient mechanism.

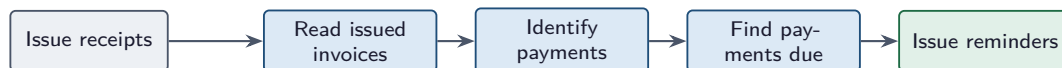
Name	Repository
Description	All data in a system is managed in a central repository accessible to all system components. Components do not interact directly, only through the repository
Example	An IDE whose components use a repository of system design information. Each tool generates information that is then available for use by other tools
When used	When you have a system in which large volumes of information are generated that must be stored for a long time. Also in data-driven systems where the inclusion of data in the repository triggers an action or tool
Advantages	Components can be independent — they need not know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently, since it is all in one place
Disadvantages	The repository is a single point of failure, so problems in it affect the whole system. There may be inefficiencies in organising all communication through the repository. Distributing the repository across several computers may be difficult



Tools do *not* communicate with each other.
All interaction is through the repository.

Figure 6.4: A repository architecture for an integrated development environment.

6.2.5 Pattern 5 — Pipe and filter



Each *filter* performs one type of data transformation; data flows through the *pipes* between them.

Figure 6.5: Pipe and filter, applied to an invoice-processing system.

Name	Pipe and filter
Description	The processing of data is organised so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows, as through a pipe, from one component to another
Example	A pipe-and-filter system used for processing invoices
When used	Commonly used in data-processing applications, both batch and transaction based, where inputs are processed in separate stages to generate related outputs
Advantages	Easy to understand and supports transformation reuse. The workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or a concurrent system
Disadvantages	The format for data transfer must be agreed between communicating transformations, and each transformation must parse its input and unparse its output to the agreed form. This increases overhead and may make it impossible to reuse transformations that use incompatible data structures. Not suitable for interactive systems

6.3 Microservices

Microservices

An architectural style in which each feature of a system runs as its own small, independently deployable service — with its own code, data store and deployment pipeline. Services communicate through APIs and can be scaled independently.

Traditional monolithic portal	Microservices portal
All features deployed together	Registration, Finance and Grades are separate services
Scale everything even if only fees are busy	Scale only the Finance service at the fee deadline
One bug can crash the whole portal	The Booking service crashes — grades still work
One large team must coordinate everything	Small teams own their own service independently
Simpler to build, test and deploy initially	Substantial operational complexity: service discovery, distributed tracing, network failure handling, data consistency across services

Exam pointer

Microservices are the industry default for large cloud systems, but the examinable point is the *trade-off*: independent deployment and scaling are bought with distributed-systems complexity. A three-person student project should almost always be a monolith.

6.4 Application architectures

Application systems are designed to meet an organisational need, and businesses in the same sector have much in common. Generic application architectures act as templates: they help you decide on the architecture for a new system, provide a checklist of features it should include, and support the organisation of work and the assessment of components for reuse.

6.4.1 Transaction-processing systems

Transaction

A coherent sequence of operations that satisfies a goal. In database terms, a transaction is an atomic unit: either all of its operations complete, or none of them do, so the database is never left in an inconsistent state.

Transaction-processing systems are designed to process user requests for information from a database, or requests to update a database. They are usually organised as an *information system* — a layered architecture with a user interface at the top, a user-communications layer, an information-retrieval layer and finally the system database.

Why atomicity matters — the Student Portal

Two students click “Register” for the last remaining place in CS1101 at the same instant. Without an atomic transaction, both availability checks return “one place free”, both inserts succeed, and the course is over-subscribed. Wrapping the check and the insert in one transaction — with appropriate locking — is a *data design* decision that prevents a defect no amount of interface testing would reliably catch.

6.4.2 Language-processing systems

Language-processing systems translate one language into an alternative representation and, for programming languages, may also execute the resulting code. Compilers translate a programming language into machine code; natural language processing systems translate one natural language into another.

A compiler is conventionally organised as a pipeline of components:

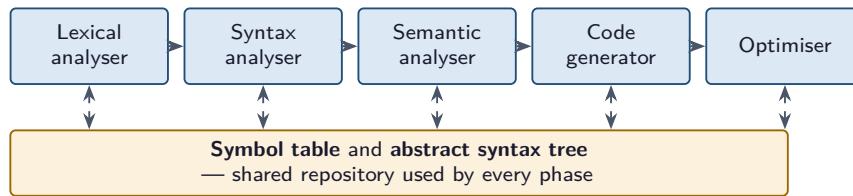


Figure 6.6: A compiler combines two patterns: *pipe and filter* for the phase sequence, and *repository* for the shared symbol table.

Key idea

Real architectures are rarely a single pattern. The compiler above composes two; the Student Portal composes layered (internally), client-server (between browser and server), MVC (within the web tier) and repository (the shared database). Recognising the composition is more useful than memorising the patterns individually.

System Modelling and UML

Learning outcomes for this chapter

- LO1.** Explain the purpose of system modelling and the four modelling perspectives.
- LO2.** Draw and interpret flowcharts using standard symbols.
- LO3.** Draw and interpret use case, class, sequence, activity, state machine, component and deployment diagrams.
- LO4.** Choose the correct diagram type for a given modelling question.
- LO5.** Avoid the notation errors that most commonly lose marks.

7.1 Why we model

System modelling

The process of developing abstract models of a system, each presenting a different view or perspective of that system. A model is a *simplification* — it leaves out detail deliberately.

Communication	A diagram conveys structure faster than paragraphs of text. Developers, clients and managers can all discuss from the same picture
Design clarity	Drawing forces you to think the design through. Ambiguities appear before any code is written, when they are cheap to fix
Documentation	Models serve as a reference for new team members and for maintenance engineers years after the original team has moved on
Code guidance	Detailed models guide implementation, telling developers what to build before they start writing code

Key idea

Modelling is abstraction, not transcription. A sequence diagram is not a line-by-line transcript of a method; a class diagram is not a header file. Draft the level of detail that answers the question you are asking.

7.1.1 Four perspectives

No single diagram shows everything. A system as complex as the Student Portal must be understood from several angles.

Perspective	The question it answers	Diagram
External	What is outside the system? Where is the boundary and what does it interact with?	Context diagram, use case diagram
Interaction	How do users and components communicate?	Use case diagram, sequence diagram
Structural	What is the system made of?	Class diagram, component diagram
Behavioural	How does it behave — how does it respond to events over time?	Activity diagram, state machine diagram

7.1.2 UML

Unified Modeling Language

The internationally accepted standard notation for modelling software systems, maintained by the Object Management Group. It provides diagram types covering structural, behavioural and interaction perspectives. UML 2.5.1 (2017) is the current version.

UML defines fourteen diagram types. This course requires six, plus flowcharts and data flow diagrams, which are not UML but appear in the same modelling role.

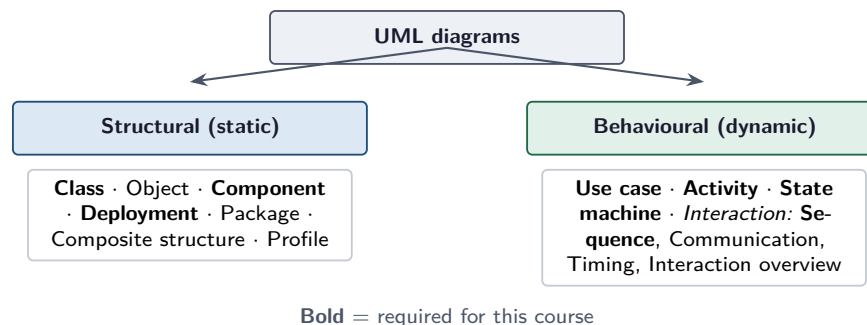


Figure 7.1: Where the six required diagrams sit within UML.

7.2 Flowcharts

Flowcharts are not part of UML, but they are the first modelling notation in the tutorial series and remain the clearest way to express the logic of a single algorithm or decision procedure.

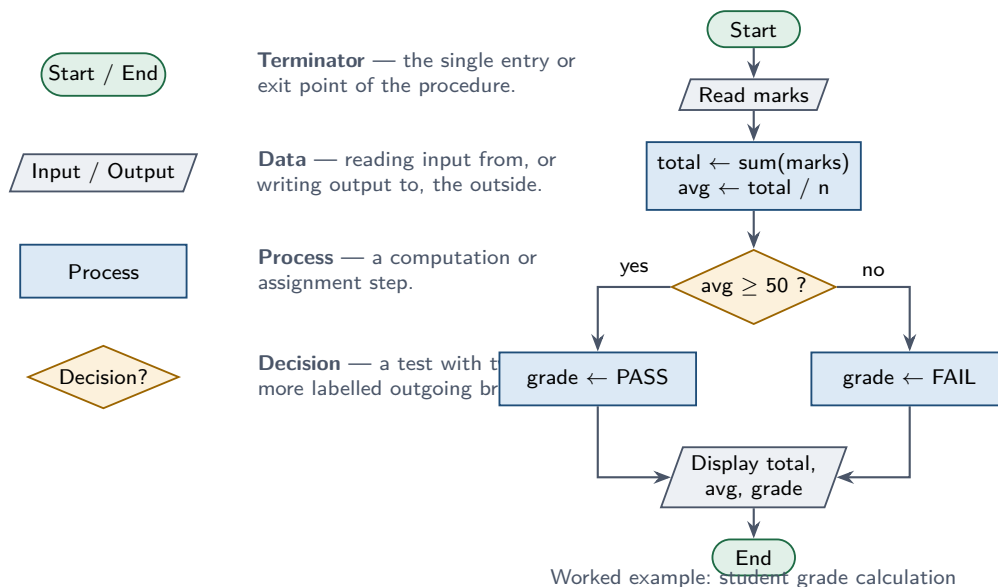


Figure 7.2: Flowchart symbols and a worked example.

Rule	Why
One entry, one exit	A procedure with several exit points is hard to reason about and hard to test
Every decision branch is labelled	An unlabelled branch is ambiguous — the reader cannot tell which path is the “yes”
Every path terminates	A path that runs off the page is an incomplete specification, not a shortcut
Loops return to a merge point	Draw the arrow back to a specific point, not into the middle of an existing arrow

Exam pointer

Tutorial 1 asks for a flowchart *before* coding for every scenario — fare calculation, billing, payroll, fine calculation, tiered electricity billing, parking fees. The pattern is always the same: read inputs, validate, branch on a condition, compute, output. Get the decision diamonds and their labels right and the rest follows.

7.3 Use case diagrams

Use case

A description of an interaction between an *actor* and the system that achieves a goal of value to that actor. A use case diagram shows the system boundary, the actors outside it, and the use cases they participate in.

7.3.1 Notation

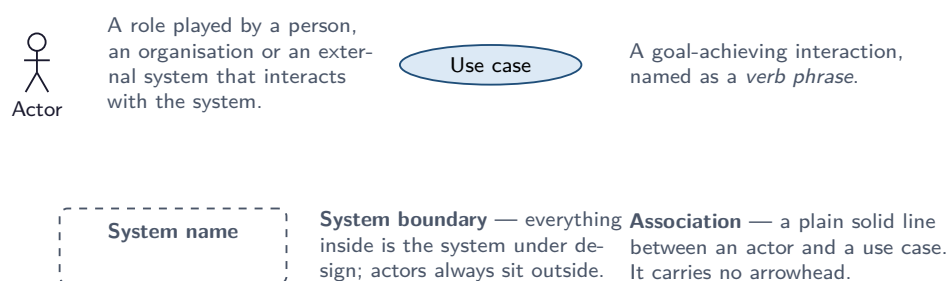


Figure 7.3: Core use case notation.

7.3.2 The three relationships between use cases

Exam pointer

The direction of the `<<extend>>` arrow is the single most common mistake in use case diagrams. Memory hook: *the arrow always points at the use case you cannot remove*. Include: the base needs the included behaviour, so base → included. Extend: the extension needs a base to attach to, so extension → base.

7.3.3 Worked example — Student Portal

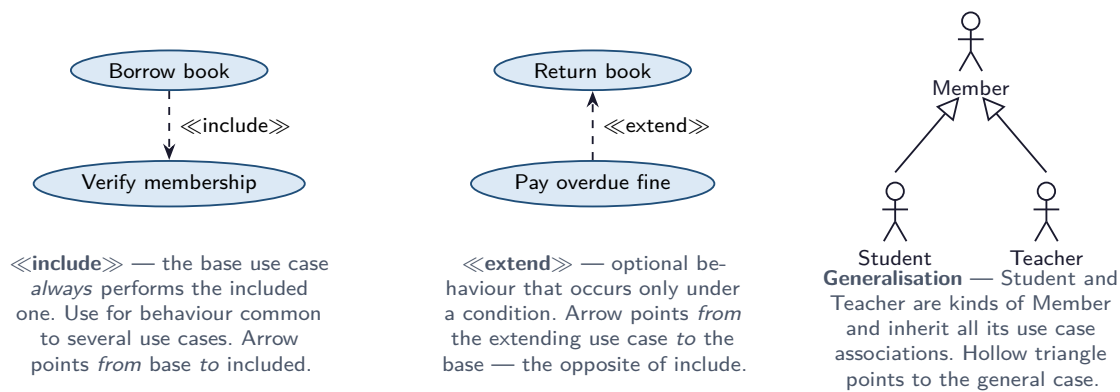


Figure 7.4: <<include>>, <<extend>> and generalisation.

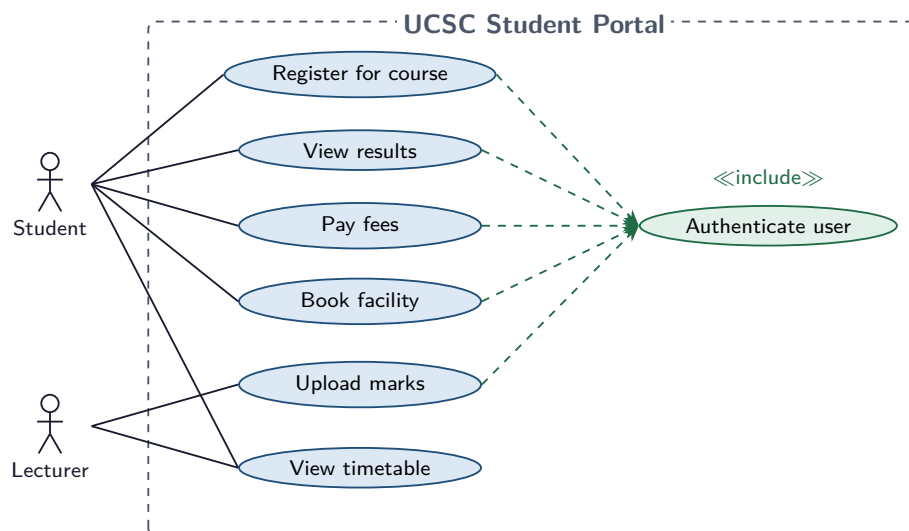


Figure 7.5: A use case diagram for the Student Portal. Authentication is factored out as a single included use case rather than repeated five times.

7.3.4 How to build a use case diagram from a scenario

1	Identify the system boundary	What are you building? Anything you are not building is an actor or is out of scope entirely
2	Identify actors	Look for <i>roles</i> in the scenario, not job titles. “Students, teachers and external visitors” are three actors — or one <i>Member</i> actor with three specialisations
3	Identify external systems	Payment gateways, email services and notification systems are actors too. They sit outside the boundary
4	Identify use cases	Look for the <i>goals</i> actors have, expressed as verb phrases: “borrow book”, not “book borrowing module”
5	Draw associations	Connect each actor to every use case they participate in, with a plain line
6	Factor common behaviour	Behaviour appearing in several use cases becomes an <<include>>. Conditional behaviour becomes an <<extend>>
7	Check for generalisation	If two actors share most associations and one is a kind of the other, generalise

Common use case diagram errors

× Decomposing into functions	A use case is a <i>goal</i> , not a step. “Enter password” is not a use case; “Log in” is
× Arrowheads on associations	The actor–use case line is a plain association. No arrow
× Actors inside the boundary	Actors are always external, by definition
× Modelling data flow	A use case diagram shows <i>who can do what</i> . It shows no sequence, no data and no conditions
× Reversed «extend»	See the exam pointer above

7.4 Class diagrams

Class diagram

A structural diagram showing the classes in a system, their attributes and operations, and the static relationships between them. It shows what the system *is*, permanently — there is no clock built into it.

7.4.1 The class box

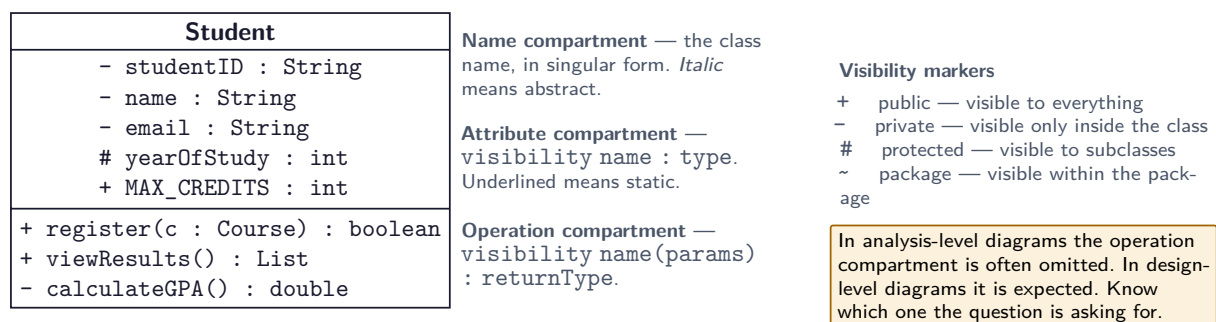


Figure 7.6: Anatomy of a class.

7.4.2 Relationships

Exam pointer

Aggregation versus composition is examined every year. The deciding question is always about *lifetime*: *if I delete the whole, must the part be deleted too?* Yes → composition (filled diamond). No → aggregation (hollow diamond). If you genuinely cannot decide, a plain association is safer than a wrong diamond.

7.4.3 Multiplicity

Notation	Meaning	Example
1	Exactly one	Each Enrolment refers to exactly 1 Course
0..1	Zero or one (optional)	A Student has 0..1 current scholarship
* or 0..*	Zero or more	A Course has 0..* Enrolments
1..*	One or more	An Order has 1..* OrderLines
2..5	A specific range	A Paper has 2..5 assigned Reviewers

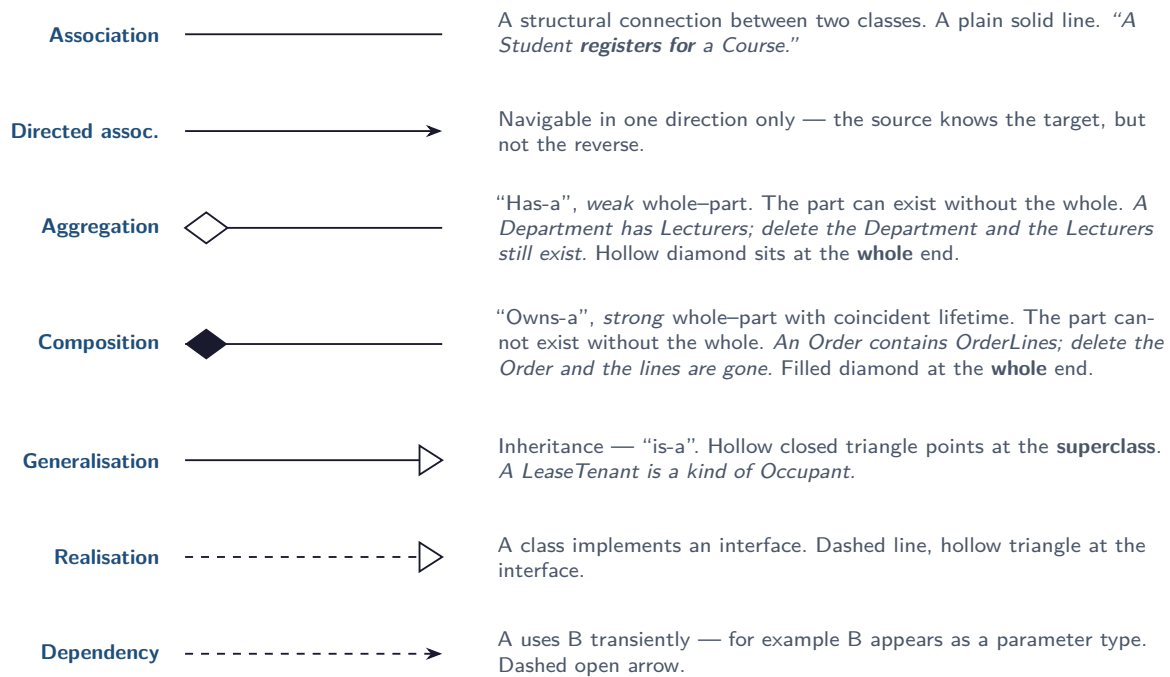


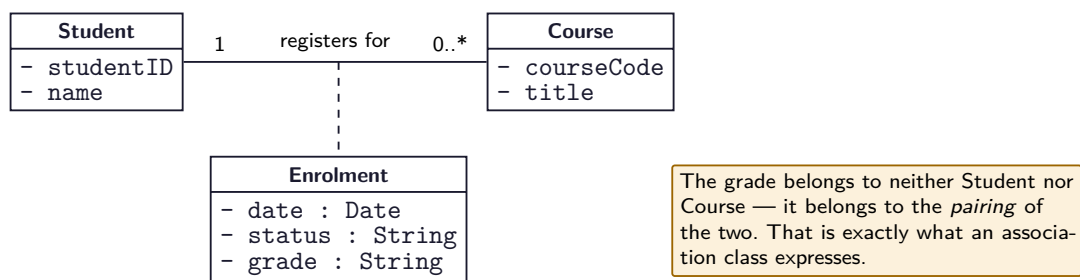
Figure 7.7: Class relationships, weakest to strongest.

Key idea

Read multiplicity *across* the association, not at the end you are standing on. On *Student 1 -- 0..* Enrolment*, the 0..* sits at the Enrolment end and means “one Student has zero or more Enrolments”.

7.4.4 Association classes

When the relationship itself carries data, model it as an *association class* — a class attached to the association by a dashed line.



The grade belongs to neither Student nor Course — it belongs to the *pairing* of the two. That is exactly what an association class expresses.

Figure 7.8: An association class.

7.4.5 Worked example — Student Portal

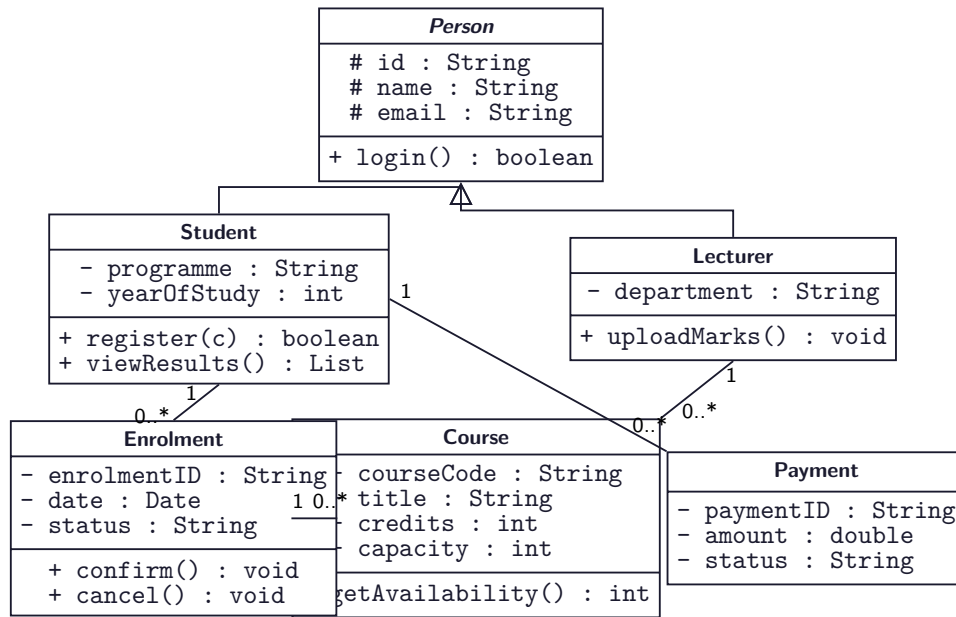


Figure 7.9: A class diagram for the Student Portal. *Person* is abstract (shown in italics); *Student* and *Lecturer* inherit from it.

7.4.6 Building a class diagram from a scenario

1	Find candidate classes	Underline the nouns in the scenario. Most classes are nouns — but not every noun is a class
2	Discard non-classes	Remove synonyms, attributes masquerading as classes (“address” is usually an attribute), and things outside the system boundary
3	Assign attributes	For each class, what data does it hold? Give each a type
4	Assign operations	What behaviour belongs to this class? Behaviour goes where the data lives
5	Set visibility	Attributes are normally - private; operations that form the interface are + public
6	Draw relationships	Association, aggregation, composition, generalisation — decide each by asking about lifetime and “is-a”
7	Add multiplicity	At <i>both</i> ends of every association. A missing multiplicity is a missing decision
8	Look for generalisation	Two classes sharing many attributes usually indicate a common superclass

Recognising relationships in tutorial wording

Scenario wording

“each property building *consists of* multiple apartment units”
 “two distinct profiles of occupants: long-term lease tenants and short-term vacation renters”
 “every prescription *consists of* multiple medication line items”
 “two types of vendor profiles: Individual Artisans and Corporate Brands”
 “a single paper can be co-authored by multiple researchers”
 “the system supports distinct evaluation templates for Full Research Papers and Workshop Papers”

Relationship

Composition — units do not exist without the building
 Generalisation — *Occupant* superclass with two subclasses
 Composition — line items are meaningless outside the prescription
 Generalisation from a *Vendor* superclass
 Association with 1..* at both ends
 Generalisation from a *Submission* superclass

7.5 Sequence diagrams

Key idea

Structure tells you what exists. It never tells you what happens. A class diagram is the floor plan — a permanent record with no clock built in. A sequence diagram is the walkthrough: read strictly top to bottom, it shows which instances talk to which, in what order, to complete one transaction from start to end.

	Class diagram	Sequence diagram
Blueprint state	Frozen	In motion
Shows	Classes, attributes, associations	Which instances talk to which, in what order
Time axis	None	Downward, strictly
UML category	Structural / static	Behavioural / dynamic

7.5.1 Lifelines and activation

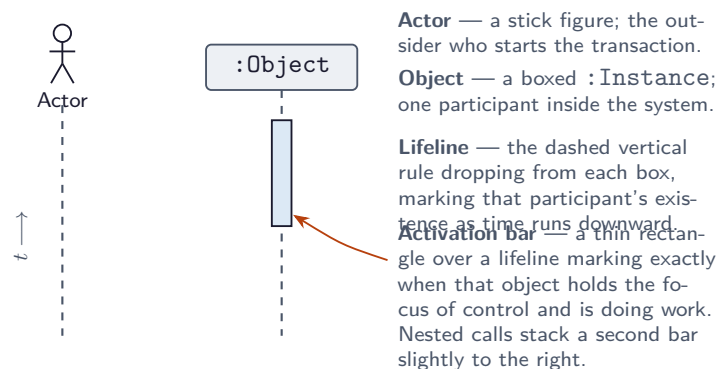


Figure 7.10: Lifelines and activation bars.

7.5.2 Creation and destruction

A lifeline need not run the full height of the diagram — it can start or stop part-way.

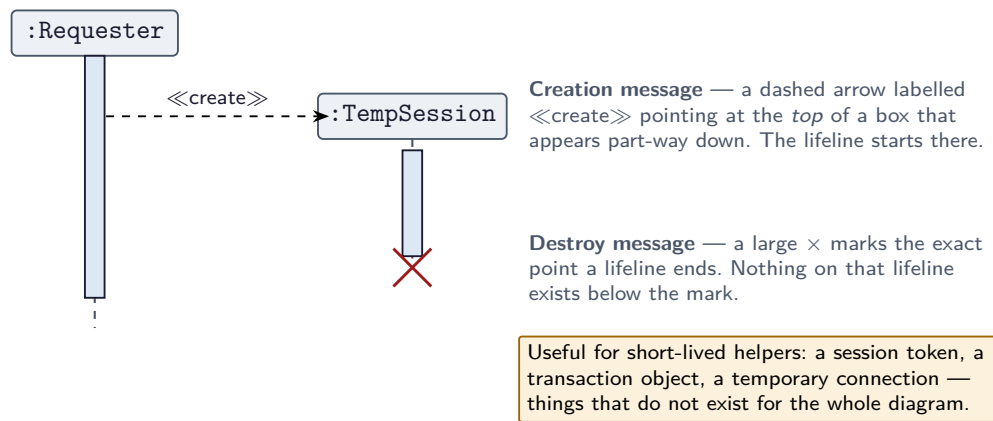


Figure 7.11: Object creation and destruction.

7.5.3 The four message types

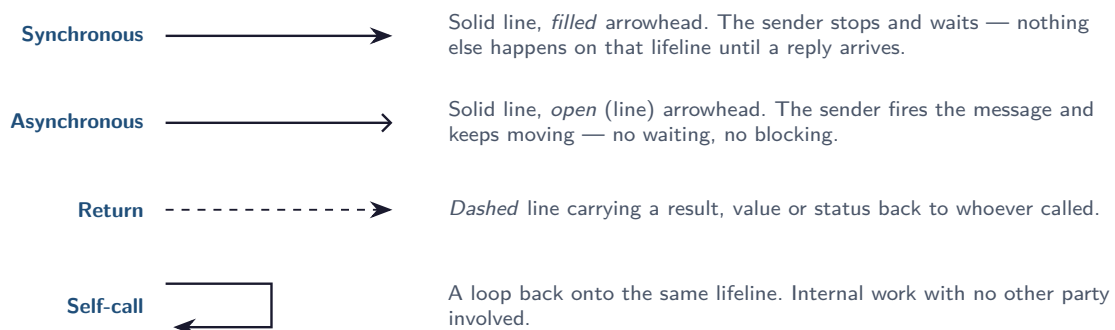


Figure 7.12: The four marks you will draw most often.

7.5.4 Three rules that govern the drawing

Activation bars mark work	A thin rectangle over a lifeline shows exactly when that object holds the focus of control. An unmarked lifeline reads as idle
Gravity runs one way	Down is later. Anything drawn higher on the page happened first. There is no other reading of the axis
Runtime, not source	You are drafting what <i>instances</i> actually do to each other at runtime — not restating the class's compiled properties

7.5.5 Worked example 1 — a login, start to finish

7.5.6 Combined fragments

Real transactions branch, repeat, run things at the same time and sometimes exit early. **Combined fragments** draw this without the diagram becoming a mess.

alt	Alternative — strict if/else. The frame is split by dashed lines into <i>operands</i> , each with its own [guard]. Exactly one operand runs
opt	Optional — a plain <i>if</i> with no else. One operand with one [guard]: it runs when the guard is true, and is simply skipped otherwise
loop	Loop — the enclosed messages repeat until a boolean guard turns false, e.g. loop [hasNext]
par	Parallel — operands execute concurrently, in no defined relative order
ref	Reference — collapses a whole interaction defined in another diagram into one frame

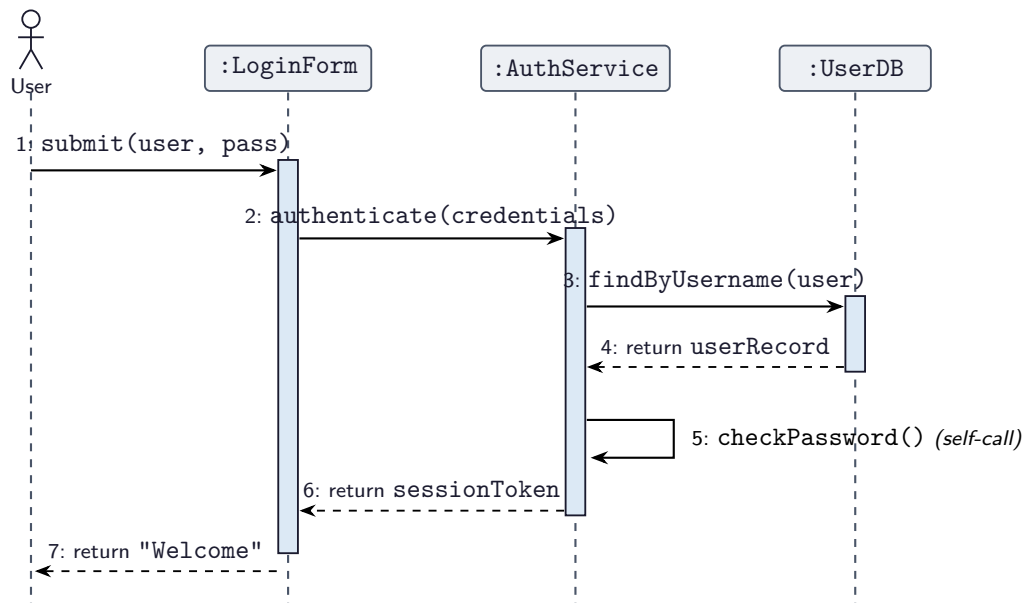


Figure 7.13: A complete login sequence — actor, objects, lifelines, activation bars and all four message types.

7.5.7 Worked example 2 — an ATM withdrawal with its branch

The transaction has four beats: the customer inserts a card and enters a PIN; the ATM sends a synchronous validation call to the bank and waits; the bank checks the credentials and returns a status; and what happens next depends entirely on that status — one diagram, two possible outcomes.

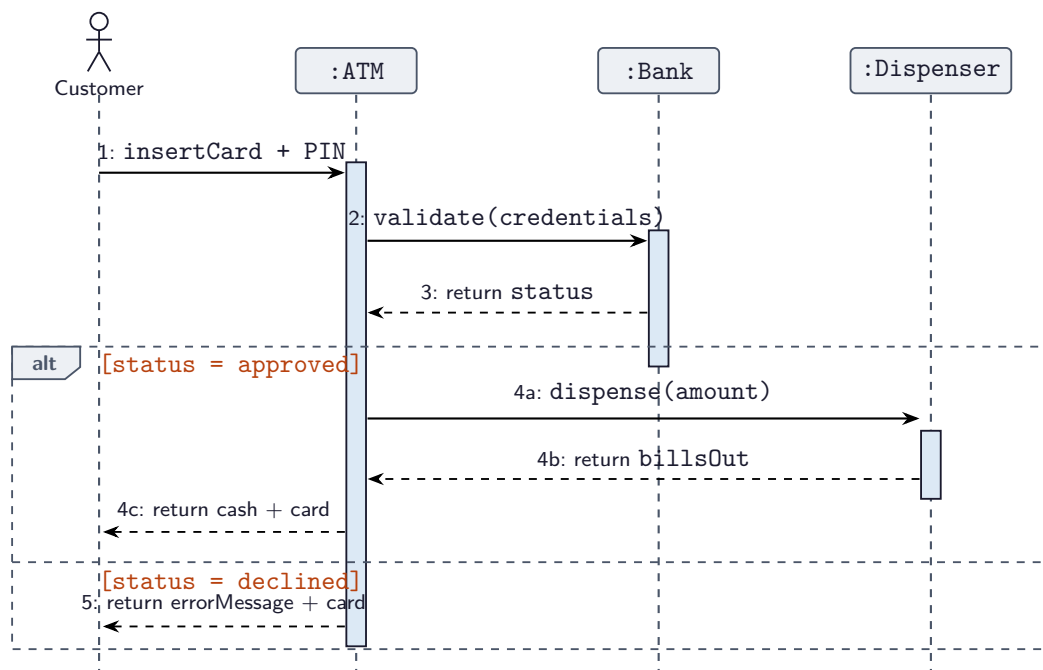


Figure 7.14: The withdrawal, with its branch. The dashed frame is a combined fragment — the notation for “one of these paths, not both”.

7.5.8 Red-pen corrections

✗ Swapped arrow styles	Solid versus dashed, filled versus open — these are not decoration. Mixing them up changes what the message means
✗ Messages travelling backward in time	A return line that slants <i>upward</i> on the page draws a contradiction. Keep every message horizontal
✗ Drafting source code, not behaviour	Getters, setters and line-by-line statements do not belong here. Stay at the level of instance-to-instance communication
✗ Forgotten activation bars	If a lifeline is doing work, that work needs a bar. An unmarked lifeline reads as idle
✗ Overlapping combined fragments	Frames nest cleanly inside one another, but they cannot cross. A partially overlapping alt and loop has no valid meaning

7.5.9 Drafting diagrams as text

Tools such as PlantUML and Mermaid render a diagram from plain syntax, which is useful for version control and quick iteration.

Minimal PlantUML for the login flow

```
@startuml
actor User
participant ":LoginForm" as LF
participant ":AuthService" as AS
participant ":UserDB" as DB

User -> LF : submit(user, pass)
activate LF
LF -> AS : authenticate(credentials)
activate AS
AS -> DB : findByUsername(user)
activate DB
DB --> AS : userRecord
deactivate DB
AS -> AS : checkPassword()
AS --> LF : sessionToken
deactivate AS
LF --> User : "Welcome"
deactivate LF
@enduml
```

7.6 Activity diagrams

Activity diagram

A behavioural diagram showing the flow of control from activity to activity — the steps of a process, where decisions occur, and which steps can run in parallel. It models a *workflow* or *algorithm*, not the collaboration of specific objects.

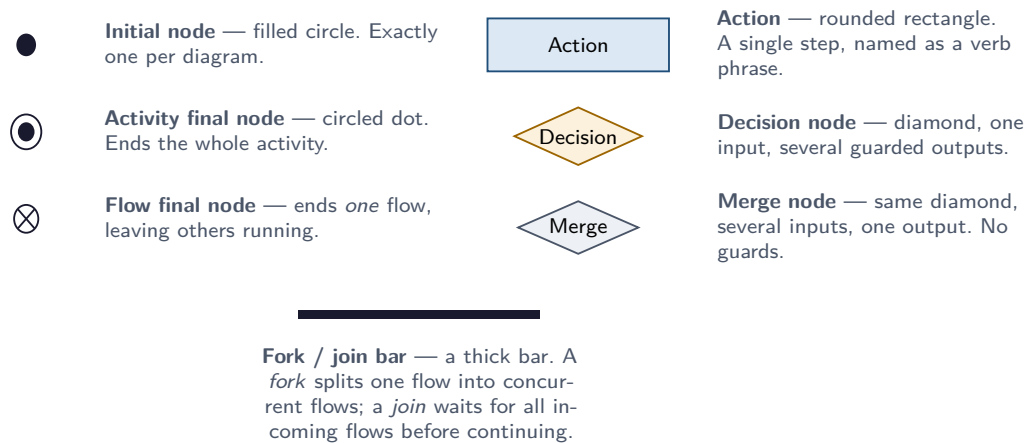


Figure 7.15: Activity diagram notation.

7.6.1 Notation

Exam pointer

Decision and merge use the *same* diamond symbol but are not interchangeable. A decision has one incoming and several outgoing edges with [guards]; a merge has several incoming and one outgoing edge with *no* guards. The same distinction holds for fork (one in, many out) and join (many in, one out) on the bar symbol. Tutorial 7 asks explicitly for fork, join and merge, so this is worth getting exactly right.

7.6.2 Worked example — applying for a library card

This is Tutorial 7, Scenario 1, modelled in full.

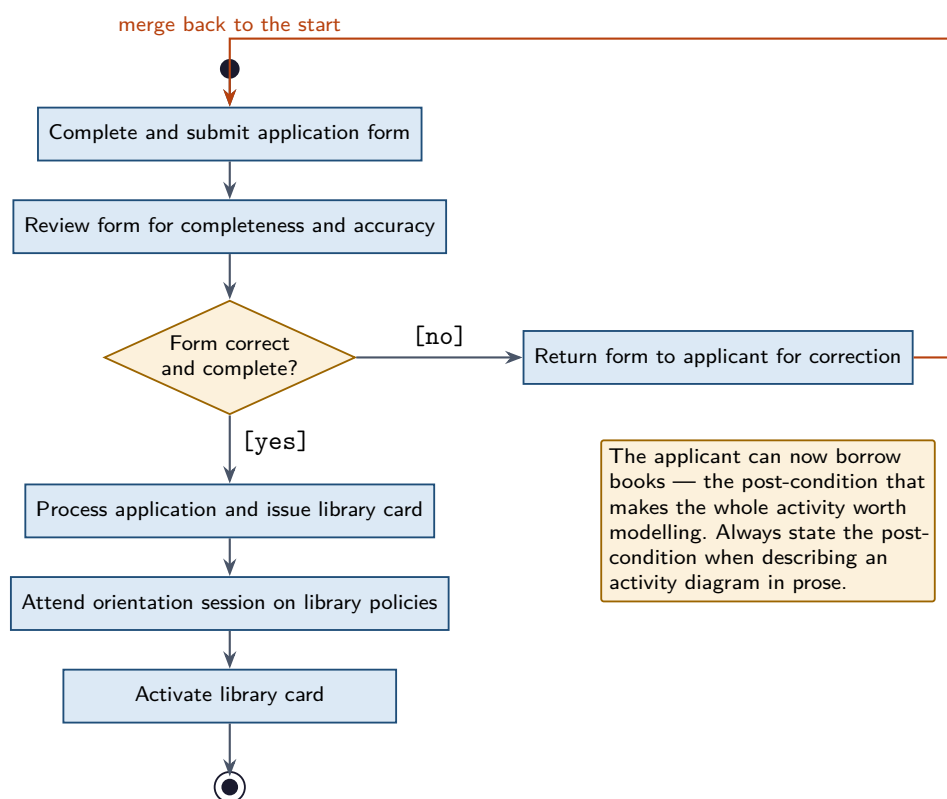


Figure 7.16: Activity diagram: applying for a library card.

7.6.3 Worked example — fork and join

Tutorial 7, Scenario 4 requires an explicit fork and join: after registration the student must pay any workshop fee *and* receive confirmation and instructions; only when both are complete does final confirmation follow.

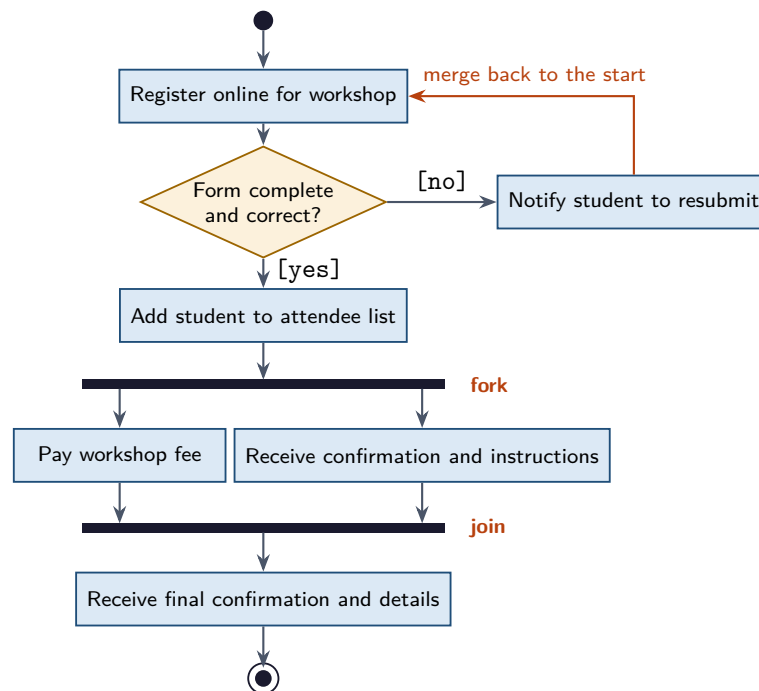


Figure 7.17: Activity diagram with a fork and join: the two branches proceed concurrently, and the join waits for both.

7.6.4 Swimlanes

Partitions (“swimlanes”) attribute each action to the actor or organisational unit responsible for it. They turn an activity diagram into a responsibility map.

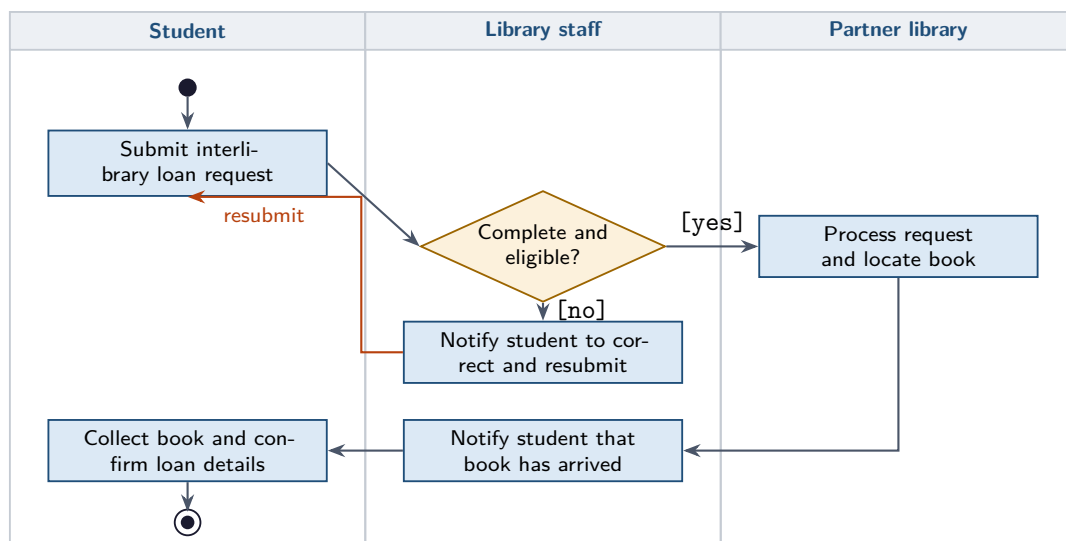
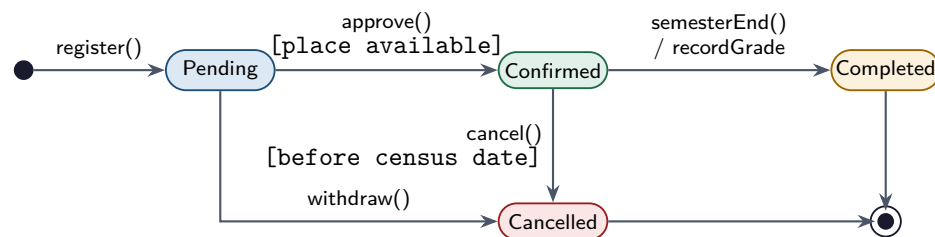


Figure 7.18: Swimlanes: Tutorial 7, Scenario 3 (interlibrary loan), with responsibility made explicit.

7.7 State machine diagrams

State machine diagram

Shows the states an object can occupy, the events that cause transitions between them, and the actions performed. Use it when one object's behaviour depends heavily on its current status.



Transition syntax: event [guard] / action.
All three parts are optional, but the order is fixed. A transition with only a guard fires as soon as the guard becomes true.

Figure 7.19: The lifecycle of an `Enrolment` object in the Student Portal.

Key idea

Reach for a state machine when the same operation behaves differently depending on status — `cancel()` on a `Pending` enrolment is not the same as `cancel()` on a `Completed` one. If every operation behaves the same regardless of status, a state machine adds nothing.

7.8 Component and deployment diagrams

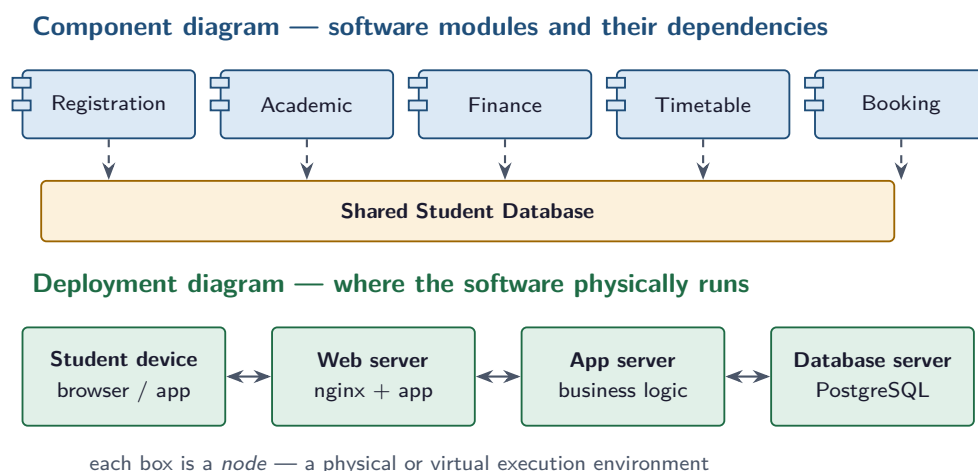


Figure 7.20: Component and deployment views of the Student Portal.

7.9 Context models and data flow diagrams

Context models define the system *boundary*: what is inside the system being built and what is an external entity it must interact with. Deciding the boundary is a political as well as a technical act — it determines which organisational units are affected.

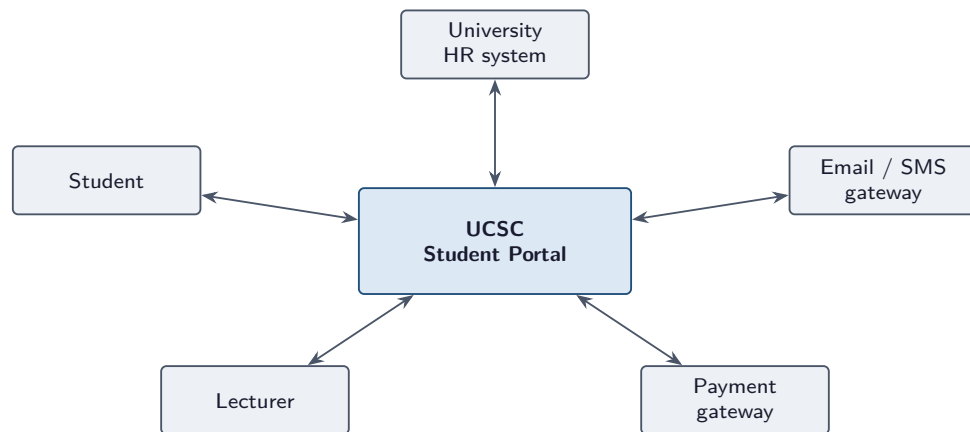


Figure 7.21: A system context model. Every arrow is an interface that must be specified.

Data flow diagrams model the *processing* perspective: how data moves through a system and is transformed at each step. They use four symbols only — external entity, process, data store and data flow — and are decomposed level by level, from a context diagram (level 0) into progressively more detailed levels.

7.10 Choosing the right diagram

Question you are asking	Diagram	Why this one
What can each user do with the system?	Use case	Shows actors and their interactions — the external view
What are the main entities and their data?	Class	Shows the static structure — things and relationships
What messages flow between objects, and in what order?	Sequence	Shows time-ordered message passing for one scenario
What are the steps in this business process?	Activity	Shows action flow including decisions, branches and concurrency
How does one object behave as its status changes?	State machine	Shows states and the events that trigger transitions
What software modules make up the system?	Component	Shows modules, their interfaces and dependencies
What hardware runs the system?	Deployment	Shows servers, devices and where software components live
What is inside the system and what is outside?	Context	Shows the boundary and every external interface
How does data move and get transformed?	Data flow	Shows processes, stores and the data flowing between them
What is the logic of this one algorithm?	Flowchart	Shows sequential steps and decision points at code level

Exam pointer

A frequent exam instruction is “choose an appropriate diagram and justify your choice”. The justification carries the marks. Name the *perspective* the diagram serves (external, interaction, structural, behavioural) and state what the alternative diagrams would have failed to show.

Software Implementation

Learning outcomes for this chapter

- LO1.** Explain the role of implementation in the lifecycle and its relationship to design.
- LO2.** Identify the major programming paradigms and apply appropriate criteria to select a language.
- LO3.** Describe the professional development environment — IDEs, package managers, linters and formatters.
- LO4.** Apply good coding practices — reliability, readability, reuse, and the SOLID principles.
- LO5.** Conduct and participate in a code review — walkthrough, inspection and pull-request workflows.
- LO6.** Use Git for version control — commit, branch, merge, push, pull, and collaborate via GitHub.
- LO7.** Explain open-source licensing and identify the common licence models and their implications.
- LO8.** Describe how AI tools change the implementation paradigm and what responsibilities engineers retain.

8.1 What is implementation?

Implementation

The process of translating a software design specification into executable source code. It bridges the gap between what a system *should* do (design) and a working system that actually *does* it.

Implementation must answer three questions:

Which language?	Choose the most appropriate programming language for the problem, the team and the environment (§8.2)
How to write it?	Apply good coding practices — readable, reliable, maintainable, reusable (§8.4)
How to manage it?	Use tools — version control, IDEs, testing frameworks — to keep quality high as the team grows (§8.3, §8.6)

8.1.1 Implementation in the lifecycle

8.1.2 A misconception about implementation

Key idea

Software engineering is not the same as writing code. Coding is a small fraction of total project cost — but the *quality* of that code determines the cost of everything that comes after it.

Exam pointer

A common short-answer question asks why coding is “not the most important activity”. The expected reasoning: coding consumes only 15–20% of development effort, but code quality

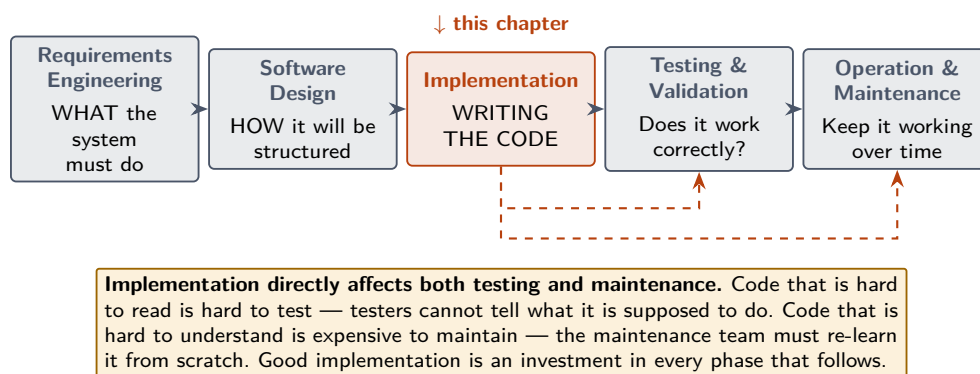


Figure 8.1: Implementation sits between design and testing, and its quality propagates forward into both.

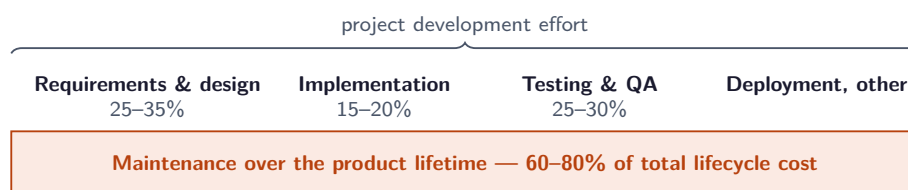


Figure 8.2: Where effort goes. Implementation is the smallest development activity; maintenance dwarfs everything (see also §1.3).

determines testing cost and the 60–80% of lifecycle cost spent on maintenance. Quality at implementation time is leverage over every later phase.

8.2 Programming languages

8.2.1 How languages evolved

Gen.	Kind and era	Character
1GL	Machine code 1940s–50s	Binary instructions — the only language a computer understands natively. Impossible for humans to write reliably at scale
2GL	Assembly language 1950s–60s	Human-readable codes (ADD, MOV, JMP) mapping directly to machine instructions. Faster to write than binary, but tied to specific hardware
3GL	High-level languages 1960s–present	C, Pascal, Java, Python — hardware-independent, structured programming. One line compiles to many machine instructions
4GL	Domain-specific languages 1970s–present	SQL for databases, MATLAB for mathematics, R for statistics. Very productive within one domain, limited outside it
5GL	Declarative and AI languages 1980s–present	You describe <i>what</i> you want, not <i>how</i> to compute it. Lisp, Prolog — and increasingly, natural-language prompts to AI systems

8.2.2 Programming paradigms

Paradigm

A style or way of thinking about programming. Most modern languages support *several* paradigms rather than exactly one.

Paradigm	Core idea	Languages and use
Procedural / structured	Write a sequence of instructions; use functions to organise and reuse code; execute step by step	C, Pascal, early BASIC. Still widely used in scripting and systems programming
Object-oriented	Organise code as objects — data plus the methods acting on it. Encapsulation, inheritance, polymorphism	Java, C++, C#, Python. Dominant in enterprise systems, Android, games
Functional	Functions are the primary building block; avoid shared state and mutable data; pure functions, no side effects	Haskell, Erlang, Scala; features in Python and JavaScript. Data pipelines, concurrency, finance
Declarative	Describe <i>what</i> result you want, not <i>how</i> to compute it; the runtime decides how	SQL, HTML, React. Growing with AI prompting — you describe the outcome

Key idea

Object-oriented design (§5.4) and the class diagrams of Chapter 7 assume the OO paradigm. The paradigm you choose at implementation time should match the paradigm you designed in — a class model implemented procedurally loses the encapsulation that justified it.

8.2.3 The modern landscape

Language	Where it is used and why
Python	General purpose, data science, AI/ML, scripting, rapid prototyping. Beginner-friendly, enormous library ecosystem (pip)
JavaScript / TypeScript	Web development, front-end and back-end. Runs in every browser; TypeScript adds strong typing; Node.js for server-side
Java / Kotlin	Enterprise applications, Android. Strongly typed, JVM ecosystem. Kotlin is Java's cleaner, safer successor
C / C++	Systems programming, embedded systems, game engines, operating systems. Closest to hardware; fast, but manual memory management
C#	Microsoft ecosystem, game development (Unity), enterprise .NET
Go	Cloud infrastructure, microservices, network tools. Fast to compile, concurrent by design (Kubernetes, Docker)
Rust	Systems programming where safety matters — prevents memory bugs at compile time. Growing for security-critical code
SQL	Every application that stores data uses it. Essential literacy for every engineer, not merely a query language

8.2.4 Choosing a language

Technical factors	Non-technical factors — often decisive
Domain fit — is the language designed for this kind of problem?	Team expertise — use what the team already knows; retraining costs time and money
Performance — does it need raw speed? C/C++/Rust for speed-critical work	Timescales — a familiar language with good tooling beats a perfect language learned under deadline
Safety and security — safety-critical systems need strong typing and memory-safety guarantees	Maintenance — the maintenance team must understand the language years from now
Platform and interfaces — mobile, web, embedded? Must it interface with existing systems?	Organisational standards — many organisations standardise on one or two languages
Library ecosystem — are the libraries you need available? pip/npm/Maven availability matters enormously	Morale — developers who enjoy the language are more productive, but this should not outweigh other factors on critical projects

Key idea

No language is right for everything. The best language is usually the one the team knows well, with good libraries for the domain.

Exam pointer

“Recommend a language and justify” questions are marked on the *balance* of technical and non-technical factors. A strong answer names the domain fit *and* concedes a non-technical constraint — typically team expertise or maintenance — rather than listing language features.

8.3 The development environment

Development environment

The complete set of tools a developer uses to write, test, manage and deploy software. Choosing and configuring good tools makes the whole team faster and the code better.

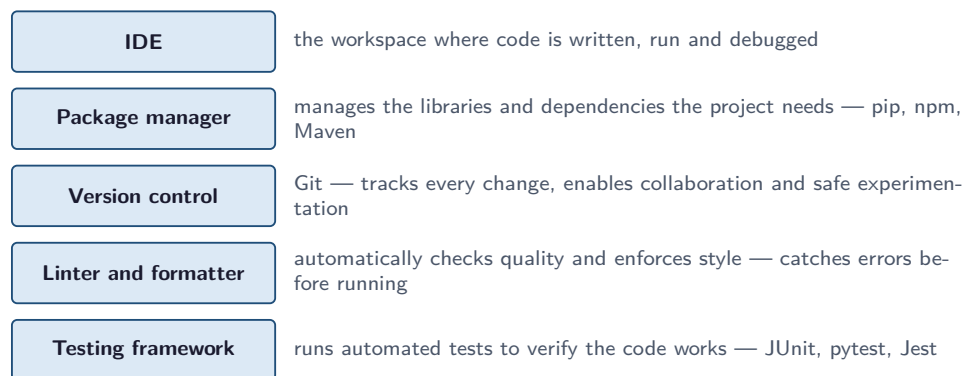


Figure 8.3: The five layers of a modern development environment.

8.3.1 Integrated development environments

Key idea

An IDE *understands the language*: it can autocomplete, detect errors before the code runs, navigate between files, and drive a debugger. A text editor is just a word processor for code.

IDE	Character
VS Code	Microsoft, free, works with any language via extensions. The most widely used IDE; built-in Git, terminal, debugger and AI (Copilot)
IntelliJ IDEA	JetBrains. The standard for Java and Kotlin; deep code analysis; used in most enterprise Java work
PyCharm	JetBrains. Strongest Python IDE — data-science tooling, virtual environments, Jupyter support
Xcode	Apple. Required for iOS and macOS development; Swift and Objective-C
Visual Studio	Microsoft. Powerful for C# and C++; enterprise-scale Windows and .NET

8.3.2 Package managers

Dependency

A library or package your code relies on — code someone else wrote that you include in your project. A package manager downloads, installs and updates these automatically.

Without a package manager

Manually download every library
Track versions yourself
Tell every new developer to install each one

One wrong version breaks everything

With a package manager

One command installs everything
Version numbers recorded in a file
New developers run one command to set up the whole project
Versions are pinned and reproducible

Manager	Language	Dependency file
pip	Python	requirements.txt
npm	JavaScript / Node	package.json
Maven / Gradle	Java	pom.xml / build.gradle

8.3.3 Linters and formatters

Linters

Analyses code for errors, bugs and style problems *without running it*. Catches issues before they become runtime failures

ESLint (JavaScript) · Pylint / Flake8 (Python)
· Checkstyle (Java) · SonarQube (any language, deep analysis)

Code formatter

Automatically rewrites code to a consistent style — spacing, line length, bracket placement. Eliminates style debates in teams

Prettier (JavaScript, CSS, HTML) · Black (Python, opinionated, zero config) · gofmt (Go, built into the language) · clang-format (C, C++)

Key idea

Both run automatically on every save or commit, so problems surface immediately rather than hours later in a code review. This is static analysis (§2.6) applied continuously rather than as a separate gate.

8.4 Good coding practices

Key idea

Code that works is the minimum. *Good* code works, is easy to understand, is unlikely to fail unexpectedly, and can be modified by people other than its author.

1. Reliability	Is the software fault-free? Does it keep working when unexpected things happen?
2. Readability	Can someone else understand, review and modify this code without asking the author?
3. Reuse	Can components be used again, here or in future projects, without rewriting from scratch?

8.4.1 Reliability — fault avoidance and fault tolerance

Fault avoidance

write code containing no errors in the first place

Follow coding standards and naming conventions — reduces ambiguity

Use strong typing — the compiler catches type errors before the code runs

Keep functions small and focused — easier to reason about and test

Write unit tests first (TDD, §3.4) — forces you to think about behaviour before logic

Peer review and pair programming — another pair of eyes catches mistakes early

Fault tolerance

accept that errors occur; handle them gracefully

Exception handling — catch expected errors and respond, do not crash

Defensive programming — validate all inputs before use; assume inputs may be wrong

Meaningful error messages — say what failed and why, clearly

Graceful degradation — if one part fails, can the rest continue?

Logging — record errors so they can be diagnosed and fixed afterwards

Exception handling — Student Portal fee payment (Python)

```
def process_fee_payment(student_id, amount):
    try:
        student = database.get_student(student_id)      # may fail if ID invalid
        if amount <= 0:
            raise ValueError("Payment amount must be positive")
        receipt = payment_gateway.charge(student, amount) # may fail if gateway down
        database.record_payment(student_id, receipt)
        return {"status": "success", "receipt": receipt}

    except StudentNotFoundError:
        return {"status": "error", "message": "Student ID not found"}
    except ValueError as e:
        return {"status": "error", "message": str(e)}
    except PaymentGatewayError:
        return {"status": "error", "message": "Payment gateway unavailable, try later"}

    finally:
        database.close_connection() # always runs, error or not
```

Key idea

The `finally` block always runs — use it to release resources such as database connections, whether or not an error occurred.

8.4.2 Readability — naming

Key idea

You write code once. It is read dozens of times — by reviewers, testers and future maintainers. A readable program is written for the next human, not only for the computer.

Naming — the most important readability decision you make

```
# POOR naming - what is x? what is lst? what is t?
def calc(x, lst):
    t = 0
    for i in lst:
        t += i.f * x
    return t

# GOOD naming - the code explains itself
def calculate_total_fees(tax_rate, enrollments):
    total = 0
    for enrollment in enrollments:
        total += enrollment.fee * tax_rate
    return total
```

Use full, meaningful names	Never abbreviate unless the abbreviation is universally known (<code>url</code> , <code>id</code> , <code>max</code>)
Functions they <i>do</i>	say what <code>calculate_total_fees()</code> , not <code>calc()</code>
Variables they <i>are</i>	say what <code>student_enrolment_count</code> , not <code>n</code> or <code>cnt</code>
Constants	in <code>MAX_ENROLMENTS</code> , not <code>max</code> or <code>m</code>
	<code>UPPER_CASE</code>

8.4.3 Comments and self-documenting code

Key idea

The best code is self-documenting — so clearly written that comments are rarely needed to explain *what* it does. Comments explain *why*.

Comment WHY, not WHAT

```
# USELESS - restates what the code obviously does
total = total + fee      # add fee to total

# USEFUL - explains a non-obvious decision and cites its source
# Late fee is waived for the first offence per semester policy (regulation 4.2.1)
if is_first_late_payment_this_semester(student_id):
    late_fee = 0

# DOCSTRING - purpose, parameters, return value, errors
def calculate_gpa(grades: list[float], credits: list[int]) → float:
    """Calculate GPA using a credit-weighted average.

    Args:
        grades - grade points per course
        credits - corresponding credit hours
    Returns: float - weighted GPA rounded to 2 decimal places
    Raises: ValueError if grades and credits differ in length
    """
```

Comment rules

Comment **why**, not **what** — the code shows what; explain the reasoning behind non-obvious decisions
 Write a docstring for every function — purpose, parameters, return value, errors
 Keep comments up to date — an outdated comment is worse than no comment at all
 If code needs a long comment to explain what it does, the *code* needs rewriting

8.4.4 The SOLID principles

SOLID

Five design principles for object-oriented code that is easy to maintain, extend and understand. Each targets a common cause of fragile, hard-to-change code.

S	Single Responsibility	A class or function should have <i>one</i> reason to change — it should do one thing only
O	Open / Closed	Open for extension, closed for modification — add new behaviour without changing existing code
L	Liskov Substitution	Subclasses must be substitutable for their parent class without breaking the program
I	Interface Segregation	Many specific interfaces are better than one large general-purpose interface
D	Dependency Inversion	Depend on abstractions, not concrete implementations — use interfaces, not specific classes

Exam pointer

SOLID is the code-level expression of the design principles in §5.3. Map them explicitly if asked: **Single Responsibility** *is* high cohesion; **Dependency Inversion** and **Interface Segregation** *are* low coupling; **Open/Closed** depends on information hiding. Naming that link earns more than reciting the five letters.

8.4.5 Technical debt**Technical debt**

The future cost of additional work created by choosing an easy solution now instead of a better, more complete one. Like financial debt it accumulates interest — the longer it is left, the more expensive it becomes to fix.

How it accumulates	Consequence
Skipping tests	“We’ll add tests later” — later never comes. Every new feature built on untested code multiplies the debt
Copy-paste programming	Duplicated code means every fix must be made in several places — and one is always forgotten
Ignoring linter warnings	Each ignored warning is a known problem left in the code; they compound into a codebase nobody wants to touch
Hard-coding values	Magic numbers, URLs and paths placed directly in code break when the environment changes

Key idea

Refactoring — improving the internal structure of existing code without changing its behaviour — is the primary way to repay technical debt. It is an XP practice (§3.4) and the countermeasure to the structural degradation that incremental development risks (§2.5).

8.5 Code quality and review**Code review**

A process in which one or more developers examine code written by another, looking for errors, design problems and improvements before it is merged into the main codebase. It is the most cost-effective quality activity in software development.

Catches bugs early	Defects found in review cost roughly 10× less to fix than defects found after deployment
Spreads knowledge	Reviewers learn the codebase; authors learn from feedback — no single point of knowledge
Enforces standards	Naming conventions, architecture decisions and quality standards are applied consistently
Improves design	A reviewer often sees a simpler solution the author missed while deep in the problem
Builds team culture	Regular review creates craftsmanship and shared responsibility for quality

8.5.1 Walkthrough and inspection

These are the two formal techniques used in plan-driven development. They are the same instruments introduced as non-execution V&V in §2.6, seen here from the implementation side.

Code walkthrough*informal · 3–7 members · author presents*

Members receive the code days beforehand and prepare test cases

In the meeting the team traces through the code manually with those test cases

The team discusses issues found; the author listens and takes notes

Goal: discover algorithmic and logical errors — not assign blame

Output: a list of issues for the author to fix

Code inspection*formal · structured roles · checklist-driven*

Roles: **moderator, author, reviewer(s), recorder**

Code is read line by line against a defect checklist

Checklist items: uninitialised variables, array bounds, infinite loops, memory leaks

All defects are recorded by the recorder; the author does not defend the code

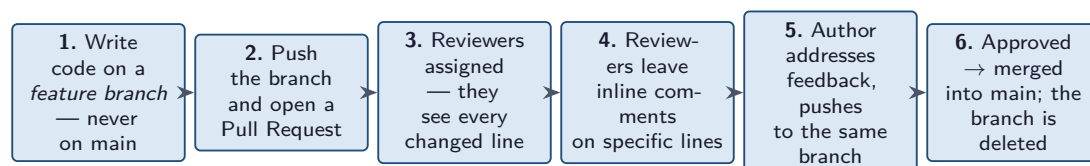
Afterwards the author fixes defects; re-inspection may be required

Key idea

Code review is not about criticising the person — it is about improving the code. Separate the code from its author.

8.5.2 Pull requests — the modern workflow**Pull request (PR)**

A formal request to merge code from one branch into another. Before merging, team members review the changes and leave comments, and the author addresses them. This is the standard code-review workflow in virtually all modern teams.



Modern practice: opening a PR also triggers automated tests and linters (CI, §2.12). The code must pass these before a human reviewer even looks at it.

Figure 8.4: The pull-request workflow.

Exam pointer

If asked to “describe how code is reviewed in practice”, give the PR workflow *and* name its plan-driven ancestors (walkthrough, inspection). The examinable insight is that a PR is an inspection with the checklist partly automated and the meeting replaced by asynchronous inline comments.

8.6 Version control with Git**Without version control — what actually happens**

“I changed the login function.” “I also changed the login function.” Whose version do we keep? What did each of us change? Who has the latest file — your email or mine? The demo is in twenty minutes and the code is broken. What did we change since yesterday?

Version control

A system that records every change made to every file over time. You can see who changed what, when and why; return to any previous version; and let multiple people work simultaneously without overwriting each other's work.

History	Every change is recorded permanently. If you break something you can go back. Nothing is ever truly lost
Collaboration	Multiple developers work on different features simultaneously; version control merges their work
Safety	Experiment freely on a branch — if it goes wrong, delete the branch. The main code is untouched

8.6.1 Git concepts

Git was created in 2005 by Linus Torvalds — the creator of Linux — because the kernel team needed a system that could handle thousands of simultaneous contributors. It is now used by virtually every software company.

Repository (repo)	The entire history of a project — every file and every change ever made — stored in a hidden <code>.git</code> folder
Commit	A saved snapshot at one point in time. Includes what changed, who changed it, when, and a message explaining why
Branch	A parallel version of the project. You branch off to work on a feature, then merge back. The main branch is protected
Remote	A copy of the repository on a server (GitHub, GitLab). Others push their changes there and pull yours
Working directory	The files on your computer right now — not yet saved to the repository until you commit

8.6.2 The three stages of a file**Key idea**

Understanding where a file is at any moment is the foundation of working with Git. Every file is in exactly one of three states.



Figure 8.5: The three stages of a file in Git.

8.6.3 Core commands

<code>git init</code>	Create a new repository in the current folder. Run once at the start of a project
<code>git clone <url></code>	Download a complete copy of a remote repository. How you start on an existing project
<code>git status</code>	Show which files are modified, staged or untracked. Run this constantly — it tells you where you are
<code>git add <file></code> / <code>git add .</code>	Move a file (or everything changed) to the staging area
<code>git commit -m 'message'</code>	Save the staged changes as a new commit. The message must explain what changed and why
<code>git push</code>	Send local commits to the remote repository. Others can now see your work
<code>git pull</code>	Download and apply the latest commits from the remote. Do this before starting new work
<code>git log</code>	See the full commit history — who, when, what message. Add <code>--oneline</code> for a compact view

8.6.4 Branching

Branch

A separate line of development. You create one to work on a feature or fix, commit there freely, then merge back into the main branch when the work is complete and reviewed.

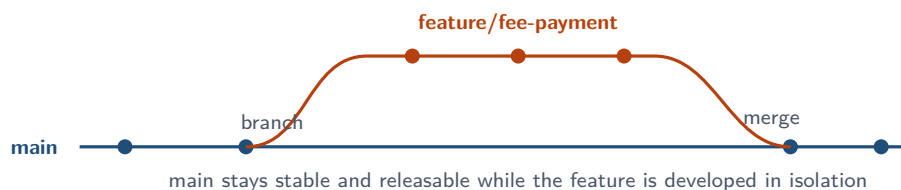


Figure 8.6: Branching: work proceeds in parallel, then merges back.

Branch commands

```
git branch                # list all branches; current marked with *
git branch feature/fee-payment # create a new branch
git checkout feature/fee-payment # switch to it (your changes now go here)
git checkout -b feature/login    # create AND switch, in one command
git merge feature/fee-payment    # merge that branch into the current one
git branch -d feature/fee-payment # delete the branch after merging
```

Branch naming conventions used in professional teams

feature/	New functionality — feature/user-login, feature/fee-payment
fix/	Bug fixes — fix/login-error, fix/null-pointer-crash
docs/	Documentation updates — docs/api-reference

8.6.5 Commit messages

Key idea

A commit message should say *why* the change was made — the code already shows what it does. Future engineers, including you, will read these messages while debugging.

Bad	Good
✗ fix	✓ Fix null pointer in <code>EnrolmentManager</code> when <code>courseCode</code> is empty
✗ changes	✓ Add late fee waiver for first offence (regulation 4.2.1)
✗ updated login	✓ Increase session timeout from 30 to 60 min — user feedback, issue #47
✗ asdfgh	✓ Refactor <code>PaymentController</code> — extract validation into a separate method

Format: a short summary of at most 72 characters saying *what* changed; then, if needed, further lines explaining *why* it was needed.

8.6.6 Git versus GitHub

Key idea

Git is the version-control tool that runs on your computer. **GitHub** is a website that hosts Git repositories online so teams can collaborate. GitLab and Bitbucket are alternatives.

Remote repository	Code stored safely in the cloud — accessible anywhere, backed up automatically
Pull requests	The standard mechanism for code review — propose a change, discuss it, get approval, merge
Issues	Track bugs, feature requests and tasks; link issues to commits and pull requests
GitHub Actions	Automate workflows — run tests on push, deploy on merge to main (CI/CD, §2.12)
GitHub Copilot	AI pair programmer built into the platform — suggests code as you type (§8.8)

`git remote add origin https://github.com/user/repo.git` connects a local repository to GitHub.

8.6.7 A complete workflow

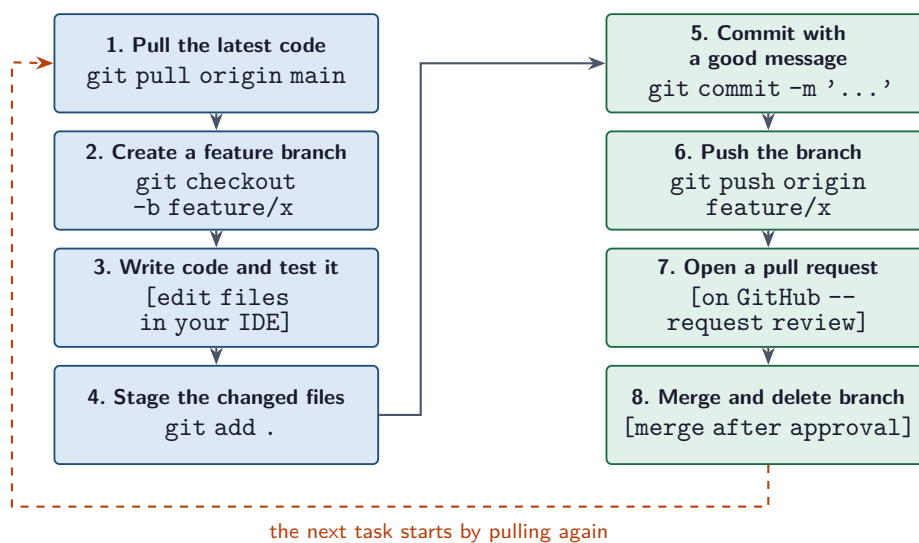


Figure 8.7: The sequence a professional developer follows for every piece of work.

8.6.8 Merge conflicts

Merge conflict

Occurs when two developers change the same part of the same file independently. Git cannot decide which version to keep, so it marks both in the file and asks you to resolve it manually.

Conflict markers in a file

```
<<<<<< HEAD                                (your version, on this branch)
def calculate_gpa(grades, credits):
    return sum(g*c for g,c in zip(grades, credits)) / sum(credits)
=====                                     (the divider)
def calculate_gpa(grades, credits, scale=4.0):
    weighted = sum(g*c for g,c in zip(grades, credits))
    return round(weighted / sum(credits), 2)
>>>>>> feature/gpa-scale                    (incoming version from the other branch)
```

- 1 Open the conflicted file — Git marks conflicts with «<<<<, ===== and >>>>»
- 2 Edit to keep the correct version — delete the markers, combining both if needed
- 3 `git add` the resolved file, then `git commit` — the merge is now complete

Key idea

Modern IDEs (VS Code, IntelliJ) show conflicts visually with *Accept Current* / *Accept Incoming* / *Accept Both* buttons — far easier than editing raw markers.

8.7 Open source and licensing

Open source software

Software whose source code is publicly available — anyone may read, modify and distribute it, subject to the conditions of its licence. The contrast is *proprietary* (closed source) software, where only the owner can see and modify the code.

You already use it	Python, VS Code, Git, Linux, Node.js, React, PostgreSQL are all open source. You cannot build modern software without it
Career infrastructure	GitHub is your professional portfolio; open-source contributions are reviewed by employers. This is your CV in code
Legal obligation	Using open-source code creates legal obligations set by its licence. Ignoring this is a real risk
Supply-chain risk	Security vulnerabilities in open-source libraries affect your software. You must track and update dependencies

8.7.1 The common licences

Licence	What it permits and requires	Used by
MIT	Very permissive. Do almost anything — use commercially, modify, distribute. Must include the original copyright notice	React, jQuery, Rails
Apache 2.0	Permissive, like MIT, but adds a patent licence protecting users from patent claims by contributors	Kubernetes, TensorFlow, Android
GPL	Copyleft — “viral”. If you distribute software using GPL code, your software must also be GPL and open source	Linux kernel, GCC
LGPL	Weaker copyleft. You may use LGPL libraries in proprietary software without opening your code, but modifications <i>to the library</i> must be shared	GNU C Library
BSD	Permissive, similar to MIT, very short. The 3-clause version adds an advertising restriction	FreeBSD, Nginx, Flask

Exam pointer

The examinable distinction is **permissive** (MIT, Apache, BSD — use freely, just attribute) versus **copyleft** (GPL — derivative work must also be open source). The practical rule: for commercial projects prefer MIT, Apache 2.0 or BSD, and evaluate any GPL dependency carefully before including it.

8.7.2 Choosing a licence for your own project

Your intention	Choose
Maximum adoption — anyone can use it for any purpose	MIT or Apache 2.0
Open, but with patent protection for users	Apache 2.0
All derivative work must remain open source	GPL — your code “stays free”
Companies may use my library, but modifications stay open	LGPL
Keep it private and own all rights	Proprietary — do not publish without legal advice

Key idea

If you publish code with *no* licence, nobody can legally use it. Always choose one. choosealicense.com is GitHub’s free guide.

8.8 AI in implementation

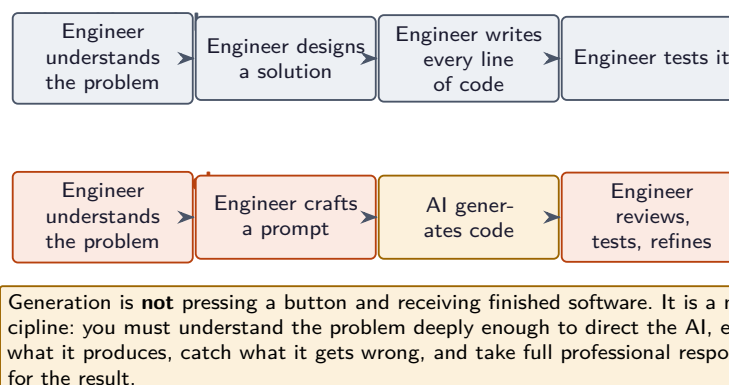


Figure 8.8: The shift from writing every line to directing and validating.

8.8.1 Skills that become *more* important

Deep problem understanding	You cannot direct an AI to solve a problem you do not understand. The engineer still defines the problem
Code reading and critical evaluation	You receive generated code. Can you read it? Does it actually do what the prompt said? Is it secure?
Testing and validation	AI-generated code can look correct and behave incorrectly. Rigorous testing becomes <i>more</i> important, not less
Architecture and design judgement	AI generates components; you design how they fit together. Architecture remains a human activity

8.8.2 The tool landscape

GitHub Copilot	Integrated into VS Code, JetBrains and others. Suggests complete functions, methods and tests as you type. Trained on billions of lines of public code
Cursor	An AI-first IDE built on VS Code. Describe what you want in natural language; it writes, edits and explains code across multiple files
ChatGPT and Claude	Conversational assistants for explaining unfamiliar code, debugging, refactoring, generating tests and writing documentation
Amazon CodeWhisperer	Particularly effective for AWS cloud development; free tier available
Tabnine	Completion trained on your team's <i>private</i> codebase — learns your project's patterns and conventions. Suited to enterprise code

8.8.3 Prompt engineering

Prompt engineering

The discipline of crafting input instructions to AI systems that produce useful, accurate and well-structured output. For code generation, a good prompt is as important as a good design specification.

Weak prompt	Effective prompt
“Write a function to check fees”	“Write a Python function <code>check_outstanding_fees(student_id: str) -> dict</code> that queries the database and returns <code>{'outstanding': bool, 'amount': float, 'due_date': str}</code> . Handle <code>StudentNotFoundError</code> with a clear error message. Follow PEP 8 style.”
<i>No language, no context, no inputs or outputs, no error handling, no style requirement</i>	<i>Specifies language, function name, signature, return type, error handling and coding standard</i>

Key idea

Write a prompt as though you are writing a requirements specification for a junior developer. Precision produces precision — and the properties of a good requirement in §4.5 apply almost unchanged.

8.8.4 What AI does well — and what it gets wrong

Use AI confidently for these

Boilerplate and scaffolding	Database connection setup, REST endpoint stubs, data model classes — repetitive structure following known patterns
Explaining unfamiliar code	Paste code you do not understand; AI explains it line by line and identifies the patterns
Generating unit tests	Given a function, AI suggests test cases including edge cases humans often miss
Refactoring suggestions	Paste messy code and ask for a more readable or more efficient version
Writing documentation	Docstrings, READMEs, API documentation — generating clear prose from structured code
Debugging assistance	Describe a bug and paste the code; AI often identifies the likely cause and explains the reasoning

Review rigorously for these

Hallucination	AI confidently calls functions that do not exist, inventing plausible-sounding API methods. Always verify library calls against real documentation
Security vulnerabilities	AI learns from all public code, including insecure code — SQL injection, XSS, insecure defaults and hardcoded credentials all appear. Always run a security linter
Wrong logic, correct syntax	Code that compiles and runs but produces the wrong result for certain inputs. Unit tests written <i>by you</i> — not by the same AI — catch this
Outdated patterns	Training data has a cutoff; generated code may use a deprecated API or miss a security patch. Check the library version
Licence issues	AI trained on open-source code may reproduce copyrighted code verbatim, creating legal risk in commercial projects (§8.7)

Key idea

AI-generated code can look completely correct and behave incorrectly. That is precisely why it must be read, understood and tested rather than trusted.

8.8.5 What the engineer still owns

You own the outcome	If generated code causes a failure, data loss or security breach, the engineer who accepted it is accountable. Responsibility does not transfer to the AI
You define the problem	AI generates solutions; you decide what problem to solve, what constraints apply, and whether the solution fits the context
You validate correctness	AI produces plausible code, not necessarily correct code. Read, understand, test and verify everything before accepting it
You make ethical decisions	AI has no professional ethics. Privacy, data protection, accessibility, fairness and social impact remain entirely with the engineer (§1.8)
You understand what you submit	Submitting code you do not understand is professional negligence, regardless of how it was generated. You must be able to explain every line you commit

Exam pointer

Every chapter of this note closes on the same professional point, and it is examined repeatedly: *AI shifts where engineering judgement is applied; it does not remove the requirement to apply it.* In implementation the sharpest formulation is the last row above — submitting code you cannot explain is negligence.

8.9 Chapter summary

L01	Implementation transforms a design specification into executable code. It is only 15–20% of development effort, but its quality determines the cost of testing and of the 60–80% of lifecycle cost spent on maintenance
L02	Paradigms: procedural, object-oriented, functional, declarative. Language choice is driven more by team expertise, timescales and maintenance than by language features
L03	The environment has five layers: IDE, package manager, version control, linter/formatter, testing framework
L04	Good code is reliable (fault avoidance + fault tolerance), readable (naming, comments explaining <i>why</i> , self-documentation) and reusable. SOLID and the avoidance of technical debt complete the picture
L05	Code review: walkthrough (informal, author presents), inspection (formal, checklist, recorder), pull request (the modern standard, with CI running first)
L06	Git: three stages (working directory, staging area, repository), core commands, branching, merging, conflict resolution, and GitHub collaboration
L07	Licences: MIT, Apache 2.0 and BSD are permissive; GPL is copyleft and viral; LGPL permits library use in proprietary code. Code with no licence cannot legally be used
L08	AI shifts the engineer from writing to directing and validating. Prompt engineering matters; hallucination, insecure code and wrong-but-plausible logic are the standing risks; responsibility stays with the engineer

Further reading for this chapter

Sommerville, *Software Engineering* 10th ed. — Ch. 3 (XP practices), Ch. 8 (testing and implementation). Pressman 9th ed. — Ch. 14–17. Martin, R. C., *Clean Code* (Prentice Hall, 2008) — the standard reference on readable, maintainable code.

Online: git-scm.com (official Git documentation) · learngitbranching.js.org (visual interactive Git tutorial) · choosealicense.com · docs.github.com/copilot · refactoring.guru/refactoring (code smells and refactoring techniques).

Software Testing

Learning outcomes for this chapter

- LO1.** Explain what software testing is, state its two goals, and place it in the V&V framework.
- LO2.** Distinguish black box from white box testing and apply the right approach at each stage.
- LO3.** Apply equivalence partitioning and boundary value analysis to design effective test cases.
- LO4.** Describe unit, component and system testing, and write automated tests with pytest or JUnit.
- LO5.** Explain Test-Driven Development and the Red–Green–Refactor cycle.
- LO6.** Describe release testing — requirements-based, scenario and performance testing.
- LO7.** Explain alpha, beta and acceptance testing, and how agile projects handle user testing.
- LO8.** Describe how AI assists test generation and what responsibilities engineers retain.

9.1 What is software testing?

Software testing

The process of executing a program with the intent of finding defects, and verifying that the program does what it is supposed to do. It is the primary activity by which we gain confidence that software is ready for use.

Goal 1 — Validation

Show that the system does what it is intended to do

Test cases are designed to *demonstrate correct behaviour* — that the system meets its requirements when used as expected

Goal 2 — Defect discovery

Discover situations where behaviour is incorrect, undesirable, or does not match the specification

Test cases are designed to *break the system* — to force failure and reveal hidden defects

Exam pointer

Every test activity serves one of these two goals, and a good answer names which. Demonstrating that a requirement is met is *validation*; hunting for the input that breaks it is *defect discovery*. A test suite containing only the first kind gives false confidence.

9.1.1 The fundamental limit

Key idea

“Testing can reveal the presence of errors — but not their absence.” (Edsger Dijkstra, 1969)

Exhaustive testing is impossible	A function with two 64-bit integer inputs has $2^{64} \times 2^{64}$ possible combinations. No team can run that many tests
Passing tests $\neq$ correct software	Tests that pass tell you the software works for the cases you <i>thought to test</i> — not for all possible inputs
Testing finds bugs, not guarantees	Afterwards you may say “we found no bugs in these scenarios”. You may not say “there are no bugs”
This is why design principles matter	Good architecture and coding practice reduce defects <i>before</i> testing begins. Testing is the last line of defence, not the first

9.1.2 Testing within verification and validation

Testing is the *dynamic* half of V&V. The framework itself — verification versus validation, and the inspection techniques that make up static V&V — is set out in §2.6; this chapter is the detailed treatment of the dynamic half.

Static V&V — inspections	Dynamic V&V — testing
Analysis of code, design or requirements <i>without running</i> the software	Executing the software with test data and observing its behaviour
Reviews, walkthroughs, static analysis tools	Unit, integration, system, release and acceptance testing
Can start before any code exists	Can only check what you can actually run
Catches defects that never execute — dead code, unreachable logic	Catches behavioural errors that inspection misses

9.1.3 The testing stages

Testing is not a single activity at the end of development — it happens throughout the lifecycle, and each stage catches a different kind of problem.

	WHO	WHAT IS TESTED	TYPICAL TOOLS
1. Unit testing	Developer	individual functions and methods in isolation	pytest · JUnit · Jest
2. Component / integration testing	Developer / QA	groups of units working together — the <i>interfaces</i>	integration frameworks
3. System testing	QA team	the complete integrated system against requirements	end-to-end tools
4. Release testing	Separate test team	the release candidate — requirements, scenarios, performance	JMeter · k6
5. User / acceptance testing	Customer / users	that the system meets real needs in the real environment	manual + automated

Figure 9.1: The five testing stages. Stages 1–3 are *development testing*, carried out by the team that builds the system; stages 4–5 are carried out independently.

Exam pointer

Note the shift in *who* tests. Development testing is done by the builders; release testing is deliberately done by a **separate team**, because familiarity with the code creates blind spots. Independence is the whole point of the separation, and questions about release testing usually turn on it.

9.2 Black box and white box testing

Key idea

Whether a test is black box or white box determines what inputs you choose, what you check, and which defects you can possibly find. You choose the approach *before* writing a single test case.

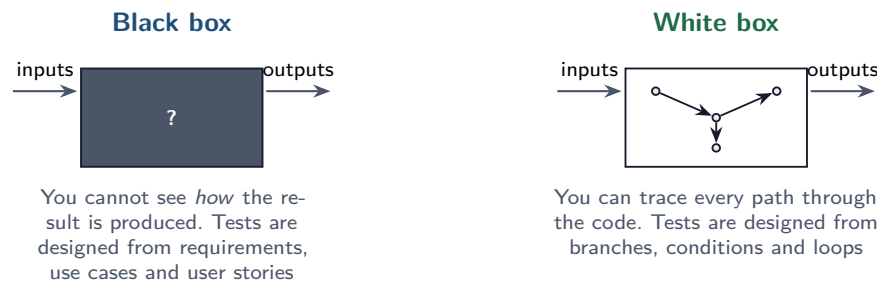


Figure 9.2: The two approaches to test design.

	Black box	White box
Tests based on	the specification	the code structure
You can see	inputs and outputs only	the actual source code
Tests derived from	requirements, use cases, user stories	logic branches, conditions, loops
Finds	missing features, wrong outputs, specification failures	logic errors, dead code, uncovered branches
Does <i>not</i> find	unused code, unreachable logic	features that were never implemented

Testing stage	Approach	Why
Unit testing	White box	The developer knows the code and tests every branch and condition
Component / integration	Both	Interfaces tested externally; internal logic checked internally
System testing	Black box	Tests derive from requirements; testers need not read the code
Release testing	Black box	A separate team uses specifications only — independence is essential
Acceptance testing	Black box	Users test what the system does, not how it works internally
Security testing	Both	External attack testing plus code review for vulnerabilities

Key idea

Professional test teams use both together — black box for specification conformance, white box for thoroughness.

9.3 Test design techniques

Key idea

You cannot test every possible input. You must choose a *small* set of test cases that maximises the chance of finding real defects. Systematic techniques make that choice intelligently rather than randomly.

9.3.1 Equivalence partitioning

Equivalence partition

A group of inputs for which the program is expected to behave identically. If the program handles one member of the partition correctly, it handles all of them correctly — so *one test per partition is sufficient*.

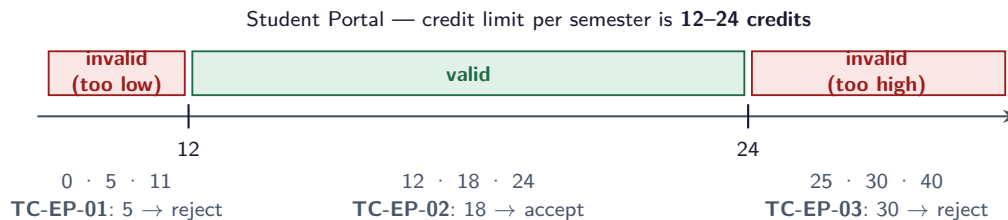


Figure 9.3: Equivalence partitioning. Testing many values from the same partition is wasted effort.

9.3.2 Boundary value analysis

Key idea

Most programming errors occur at the *boundary* of valid input ranges. Always test the exact edge values, just below the minimum, and just above the maximum.

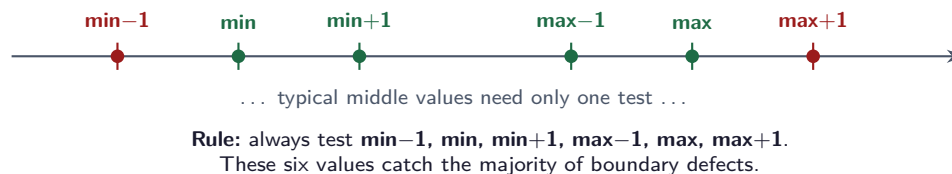


Figure 9.4: Boundary value analysis.

Boundary analysis — Student Portal GPA display

The GPA section is shown only when a student has completed *one or more* courses.

Input	Boundary role	Expected
0 courses	just below minimum	GPA section <i>not</i> displayed
1 course	minimum boundary	GPA section <i>is</i> displayed
2 courses	just above minimum	GPA section <i>is</i> displayed
10 courses	typical middle value	GPA section <i>is</i> displayed
max courses	maximum boundary	GPA section <i>is</i> displayed
max + 1	just above maximum	error: exceeded maximum enrolment

9.3.3 Testing guidelines

Guidelines encapsulate accumulated knowledge of which test cases tend to find errors. Sommerville gives these for sequences, arrays and lists:

1. Test with sequences that have only a **single value**. Programmers naturally think of sequences as containing several values and sometimes embed that assumption.
2. Use sequences of **different sizes** in different tests. This reduces the chance that a defective program accidentally produces correct output.

- Derive tests that access the **first, middle and last** elements — this reveals problems at partition boundaries.

More general guidelines (after Whittaker): choose inputs that force the system to generate all its error messages; design inputs that cause buffers to overflow; repeat the same input many times; force invalid outputs to be generated; force computation results to be too large or too small.

9.3.4 Use-case based testing

Each use case (§7.3) represents a realistic interaction, so deriving tests from use cases exercises complete, meaningful workflows rather than isolated functions.

ID	Case	Scenario — “Student registers for a course”	Result
TC-UC-01	Normal flow	Student with 15 credits selects available CS1101; expects confirmation and updated timetable	PASS
TC-UC-02	Course full	Student selects CS1101 with no remaining seats; expects a clear “course full” message	PASS
TC-UC-03	Timetable clash	Student selects CS1203, which clashes with registered CS1101; expects a clash warning	DEFECT
TC-UC-04	Prerequisite not met	Year 1 student attempts a Year 3 course; expects prerequisite rejection	PASS
TC-UC-05	Credit limit exceeded	Student adds a 4-credit course while already at 24 credits; expects a credit-limit error	PASS

9.3.5 The structure of a test case

Key idea

Without this structure, testing is not repeatable — and a test that cannot be repeated cannot be used for regression.

ID	Description	Preconditions	Steps	Expected result	Status
TC-LOGIN-01	Valid login succeeds	User registered; system running	1 open login page · 2 enter valid ID and password · 3 click Sign In	Home page displayed; student name in header	PASS
TC-LOGIN-02	Wrong password rejected	User registered	1 open login page · 2 enter valid ID, wrong password · 3 click Sign In	Error message: “Invalid credentials”	PASS
TC-LOGIN-03	Account locked after 5 failures	User account exists	1 attempt login with wrong password five times	Account-locked message; login disabled	FAIL

9.4 Development testing

Development testing

All testing carried out by the development team while building the system, before any release to external users. It comprises unit, component/integration and system testing — each at a different level of the system.

9.4.1 Unit testing

Unit

The smallest independently testable piece of software — a single function, method or class. A unit test checks that the unit does what its specification says, ignoring everything around it.

Fast	Runs in milliseconds. You run hundreds every time you save a file; slow tests do not get run
Isolated	Tests <i>one</i> thing. No database, no network, no file system — replace those with test doubles (mocks)
Repeatable	Same result every time. Does not depend on test order or on the state left by other tests
Automated	Runs without human intervention; integrated into the build so every commit triggers all tests
Self-checking	The test reports pass or fail itself — no manual inspection of output

Unit tests with pytest — Student Portal fee calculation

```
# file: test_fee_calculator.py          run with: pytest test_fee_calculator.py -v
import pytest
from fee_calculator import calculate_outstanding_fees

# --- normal cases ---
def test_student_with_no_outstanding_fees():
    result = calculate_outstanding_fees(student_id="S001", paid=50000, total=50000)
    assert result["outstanding"] == False
    assert result["amount"] == 0.0

def test_student_with_partial_payment():
    result = calculate_outstanding_fees(student_id="S002", paid=30000, total=50000)
    assert result["outstanding"] == True
    assert result["amount"] == 20000.0

# --- edge cases ---
def test_zero_total_fee_raises_error():
    with pytest.raises(ValueError, match="Total fee cannot be zero"):
        calculate_outstanding_fees(student_id="S003", paid=0, total=0)

def test_student_id_not_found_raises_error():
    with pytest.raises(StudentNotFoundError):
        calculate_outstanding_fees(student_id="INVALID", paid=0, total=50000)
```

The same tests in JUnit 5 — Java

```
// FeeCalculatorTest.java          run with: mvn test or gradle test
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

class FeeCalculatorTest {

    @Test
    void studentWithNoOutstandingFees_returnsFalse() {
        FeeResult result = FeeCalculator.calculateOutstanding("S001", 50000, 50000);
        assertFalse(result.isOutstanding());
        assertEquals(0.0, result.getAmount());
    }

    @Test
    void studentWithPartialPayment_returnsCorrectAmount() {
        FeeResult result = FeeCalculator.calculateOutstanding("S002", 30000, 50000);
        assertTrue(result.isOutstanding());
        assertEquals(20000.0, result.getAmount());
    }

    @Test
```

```

void zeroTotalFee_throwsIllegalArgumentException() {
    assertThrows(IllegalArgumentException.class, () →
        FeeCalculator.calculateOutstanding("S003", 0, 0));
}
}

```

9.4.2 Test coverage

Test coverage

The percentage of your code that is actually executed when the test suite runs. High coverage means most of the code is being exercised; low coverage means large areas have never been run by any test.

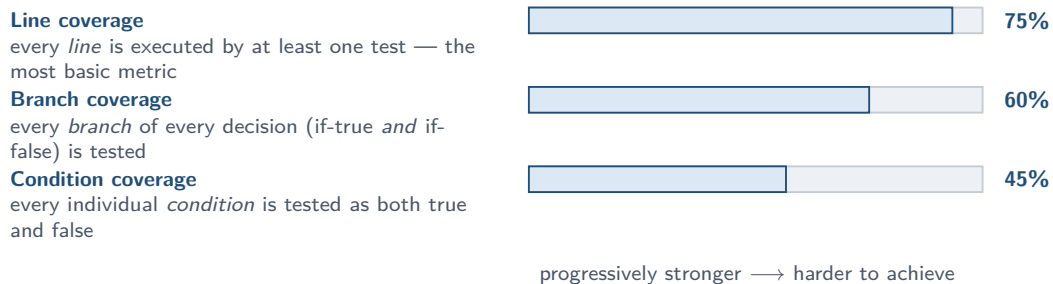


Figure 9.5: Three kinds of coverage. 80%+ line coverage is typically required in professional projects.

Exam pointer

100% coverage does not mean zero bugs. Coverage records what was *executed*, not whether it was *checked correctly* — a test that runs a line but asserts nothing still counts towards coverage. Always pair coverage with good assertions; §9.8.3 gives the technique for testing the tests themselves.

9.4.3 Component and integration testing

Integration testing

Checks that separately developed components work correctly when combined. The defects it finds are not inside individual units — they are at the *interfaces* between them: wrong data formats, timing issues, incorrect assumptions.

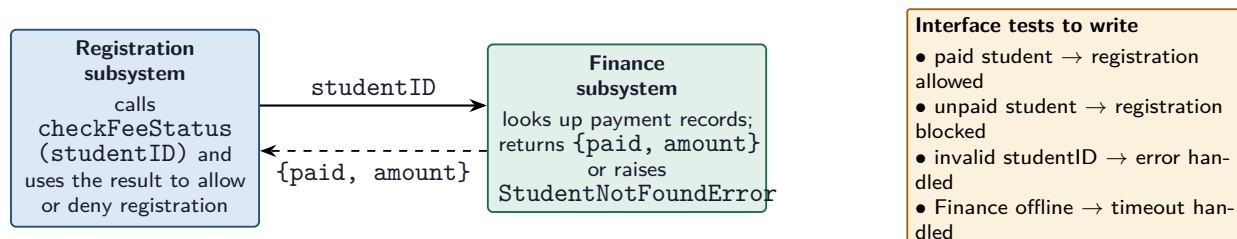


Figure 9.6: Integration testing targets the interface, not the internals.

Four kinds of interface (Sommerville)

Parameter interfaces	Data or function references passed between components. Methods have a parameter interface
Shared memory interfaces	A block of memory shared between components; one writes, others read. Common in embedded systems
Procedural interfaces	One component encapsulates a set of procedures callable by others. Objects and reusable components
Message passing interfaces	One component requests a service by passing a message and receives a reply. Client-server systems

Three classes of interface error

Interface misuse	The caller makes an error in using the interface — wrong parameter types, wrong order, wrong number
Interface misunderstanding	The caller makes wrong assumptions about behaviour — e.g. calling binary search with an unordered array
Timing errors	Caller and called component operate at different speeds and access stale data

Key idea

Interface errors are among the most common defects in complex systems, and they are invisible to unit testing — every unit can pass in isolation while the assembled system fails.

9.4.4 System testing

System testing integrates all components and tests the system as a whole. Its distinctive target is **emergent behaviour** — properties that appear only when the whole system runs together, and which therefore cannot be found at any lower level.

9.5 Test-Driven Development

Test-Driven Development

A development practice in which you write a *failing test before* writing the code it tests, then write the minimum code to make it pass. You never write production code without a failing test to justify it.

The Red–Green–Refactor cycle is introduced as an XP practice in §3.4; this section gives the full worked treatment.

TDD in practice — `calculate_gpa(grades, credits)`

```
# --- STEP 1: RED -- write the tests first; they MUST fail -----
def test_gpa_single_course():
    assert calculate_gpa([3.7], [3]) == 3.7

def test_gpa_two_courses_weighted():
    # 4.0 over 3 credits + 3.0 over 1 credit = (12.0 + 3.0) / 4 = 3.75
    assert calculate_gpa([4.0, 3.0], [3, 1]) == 3.75

def test_empty_grades_raises_error():
    with pytest.raises(ValueError):
        calculate_gpa([], [])
# run: pytest → 3 FAILED (ImportError) RED as expected

# --- STEP 2: GREEN -- minimum code to pass -----
def calculate_gpa(grades, credits):
    if not grades:
        raise ValueError("No grades provided")
    weighted_sum = sum(g * c for g, c in zip(grades, credits))
    return weighted_sum / sum(credits)
# run: pytest → 3 PASSED GREEN
```

```
# ---- STEP 3: REFACTOR -- improve without breaking -----
def calculate_gpa(grades: list[float], credits: list[int]) → float:
    """Credit-weighted GPA calculation."""
    if not grades or not credits:
        raise ValueError("Grades and credits required")
    if len(grades) != len(credits):
        raise ValueError("Length mismatch")
    weighted = sum(g * c for g, c in zip(grades, credits))
    return round(weighted / sum(credits), 2)
# run: pytest → 3 PASSED                                code is now clean
```

Benefits of TDD

Code coverage	Every piece of production code has at least one test <i>by construction</i> — you cannot write code without a test first
Fewer defects	Defects are caught within seconds of being introduced, not days later, so the cost of fixing them is almost zero
Better design	Writing tests first forces code that is <i>testable</i> — which means modular, loosely coupled, with clear interfaces
Living documentation	The test suite documents what every function is supposed to do, in a form that is always current and always executable
Regression protection	Every feature adds tests; you can change existing code confidently because the tests tell you immediately if you break something

Exam pointer

The “better design” benefit is the one most often missed in answers and is the deepest point: TDD is a *design* technique that happens to produce tests. Code that is hard to test is almost always code with high coupling — so TDD applies pressure towards the design principles of §5.3 from the opposite direction.

9.5.1 Regression testing

Regression testing

Re-running all existing tests after every change, to verify that the change has not broken anything that previously worked. “Regression” means going backwards.

Manual regression is unaffordable

If the portal has 500 test cases and you fix one bug, running all 500 manually takes days — so nobody does it, and defects slip through

Automated regression is routine

The same 500 automated tests run in two minutes on every push. Every defect introduced is caught before it reaches anyone else

Key idea

TDD naturally builds a regression suite. Every test written becomes part of a growing safety net that protects all future changes — which is why the two practices are almost always adopted together.

9.6 Release testing

Release testing

Testing a complete version of the system that is intended for use outside the development team. The goal is to convince developers, customers and the organisation that the system is good enough to release.

Who performs it?	A separate test team with no involvement in the original development. Independence matters — familiarity with the code creates blind spots
What is its goal?	Not to find every bug, but to <i>validate</i> that the system meets its requirements, performs acceptably and does not fail in normal use
How does it differ from system testing?	System testing <i>finds bugs</i> ; release testing <i>verifies readiness for external use</i> — validation, not defect hunting
What does it produce?	A release decision: approve for deployment, conditional approval with known issues documented, or reject for rework

9.6.1 Requirements-based testing

Key idea

For every requirement in the SRS or product backlog there must be at least one test case verifying it. **No requirement should be untested at release.** This is forward traceability (§4.12) made executable.

Req.	Requirement	Release tests
REQ-001	The system shall display a warning when a student's outstanding fees exceed LKR 5,000	TC-RT-001A: S001 owes LKR 6,000 → warning badge appears TC-RT-001B: S002 owes LKR 3,000 → no warning badge
REQ-002	Only registered students shall be permitted to register for courses in the current semester	TC-RT-002A: valid active student → registration form accessible TC-RT-002B: deregistered student → form blocked with explanation
REQ-003	Course registration shall close automatically 14 days after semester start	TC-RT-003A: system date day 15 → registration link unavailable TC-RT-003B: system date day 13 → registration link available

9.6.2 Performance and load testing

Key idea

A system can meet every functional requirement and still fail in production because it is too slow, crashes under load, or degrades unacceptably when many users arrive at once. Non-functional requirements (§4.4) are verified here or not at all.

Load testing	Test under the <i>expected</i> number of concurrent users — e.g. 5,000 students logging in on results day
Stress testing	Increase load <i>beyond</i> normal capacity until the system fails. The goal is to learn <i>how</i> it breaks — gracefully or catastrophically
Spike testing	Simulate sudden peaks — all students attempting to register the moment registration opens at 09:00 on Monday
Endurance testing	Run under normal load for an extended period. Reveals memory leaks and resource exhaustion that appear only over hours or days

Tools: Apache JMeter (free, widely used) · k6 (modern, script-based) · Locust (Python).

9.7 User testing

Key idea

The user's working environment has a major effect on reliability, performance, usability and robustness — effects that cannot be replicated in a lab. Real users do unexpected things; real devices, networks and workflows reveal defects controlled environments never surface.

Type	Where and who	Goal
Alpha testing	Selected real users at the <i>developer's</i> site, with the team present to observe	Find usability and real-world behaviour problems the team cannot see, because they know the system too well
Beta testing	A larger group of real users in <i>their own</i> environment, without the team present	Discover problems that emerge only in diverse real-world conditions, networks, devices and usage patterns
Acceptance testing	The customer, formally, against agreed criteria	Decide whether to accept delivery or require further changes

Exam pointer

Alpha and beta are defined by *whose environment* the testing happens in — the developer's site versus the user's own environment — not by how finished the software is. This pair is reversed in answers more often than any other in the subject.

9.7.1 Acceptance testing — the formal gate

Acceptance testing

A formal process in which the customer tests the delivered system against the agreed acceptance criteria. The outcome is a binary decision: accept delivery or reject it.

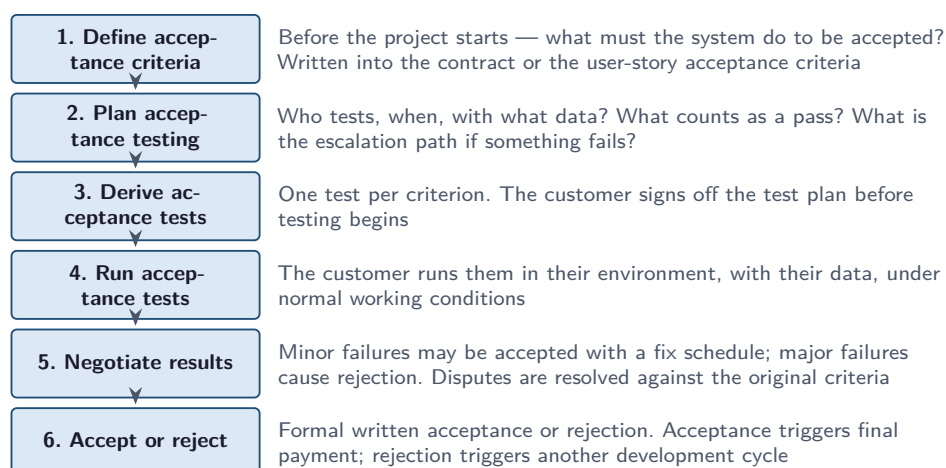


Figure 9.7: The six stages of acceptance testing.

Key idea

In agile the customer is part of the team and accepts each user story at Sprint Review (§3.3). Acceptance becomes *continuous* rather than a single final gate — which is why agile projects rarely experience the late, contractual acceptance disputes that plan-driven projects must plan for.

9.8 Modern testing practice

9.8.1 Continuous testing in CI/CD

Every push triggers the pipeline described in §2.12 and §8.6; what actually *runs* in that pipeline is primarily tests.

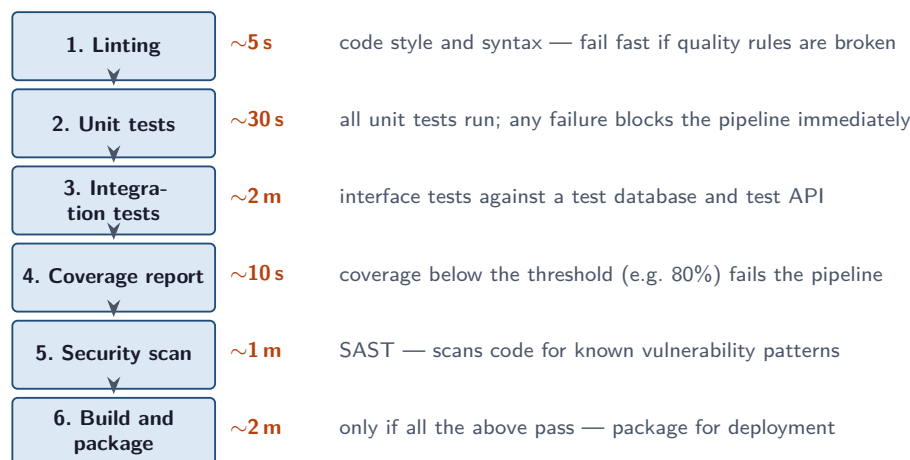


Figure 9.8: What runs automatically on every push. The ordering is deliberate: cheapest and fastest checks first, so failures surface in seconds.

9.8.2 API testing

Modern systems are built from web services, so testing the API directly — independently of the user interface — verifies server-side logic before the front end even exists.

Postman tests for POST /api/v1/registration

```
// request body: { "studentId": "S001", "courseCode": "CS1101" }

// ---- successful registration -----
pm.test("Status code is 201 Created", () => pm.response.to.have.status(201));
pm.test("Response contains confirmation", () => {
  const body = pm.response.json();
  pm.expect(body.status).to.equal("confirmed");
  pm.expect(body.courseCode).to.equal("CS1101");
});

// ---- course already full -----
pm.test("Full course returns 409 Conflict", () => pm.response.to.have.status(409));
pm.test("Error message is meaningful", () => {
  pm.expect(pm.response.json().message).to.include("no remaining seats");
});
```

Postman	GUI tool for designing, sending and testing API requests — the industry standard
REST Assured	Java library for API testing, used inside JUnit test suites
pytest + httpx	Python API testing — send HTTP requests inside pytest tests

9.8.3 Mutation testing — testing the tests

Mutation testing

A technique that introduces deliberate, tiny defects (*mutations*) into the source code and checks whether the test suite detects them. If the tests still pass when the code contains a bug, the tests are not checking thoroughly enough.

Mutation killed

A test fails when the mutation is introduced — the suite is strong enough for that code path

Mutation survived

All tests still pass — the suite does not check this behaviour, and a real bug of the same shape would go unnoticed

A surviving mutation reveals a weak test

```
# original
def is_fee_overdue(amount, days_late):
    return amount > 0 and days_late > 30

# the existing test suite
def test_fee_overdue():
    assert is_fee_overdue(1000, 31) == True
    assert is_fee_overdue(0, 45) == False

# MUTATION: change > to >=
    return amount > 0 and days_late >= 30
# does the suite catch it? 31 >= 30 is still True, 0 still fails the first
# clause -- both assertions still pass. MUTATION SURVIVES: the test is weak.

# FIX: add the boundary case the mutation exposed
    assert is_fee_overdue(1000, 30) == False
```

Key idea

Mutation testing is the answer to “how do I know my tests are any good?” Coverage tells you what was *executed*; the mutation score tells you what was actually *checked*. Tools: PIT (Java), mutmut (Python), Stryker (JavaScript). Aim for above 80% on critical business logic.

Exam pointer

Notice that the surviving mutation above is exactly a boundary defect — the technique of §9.3 would have caught it during test design. Mutation testing is best understood as an *audit* that finds where systematic test design was not applied.

9.9 AI in software testing

Key idea

Writing comprehensive test cases manually is slow and error-prone; engineers miss edge cases they did not think to consider. AI can generate test cases from code or specifications in seconds, covering scenarios human testers commonly overlook.

Test case generation	Given a function or API specification, AI generates a full set of cases — happy paths, edge cases and error conditions — faster than a human can
Code-to-test generation	Copilot, Diffblue Cover and similar tools analyse existing source and write unit tests with assertions. Point at a function, get a test suite
Bug prediction	Models trained on large repositories predict which areas are most likely to contain defects, focusing manual effort where it matters
Anomaly detection	AI monitors running systems and flags behaviour deviating from the learned normal pattern — catching defects in production before users report them
Visual regression testing	AI compares screenshots before and after a UI change and flags unexpected differences, including subtle layout shifts

What AI gets wrong — generated tests need review

Tests that always pass	Assertions that are true for any input — <code>assert result is not None</code> — pass always and catch nothing
Missing business rules	AI generates tests from <i>code structure</i> . If the code is missing a business rule, AI cannot generate a test for it
Wrong assertions	AI may assert the wrong expected value, especially for computed results. Verify that the expected value is actually correct
Incorrect mocking	Generated mocks may not simulate real dependencies; the test passes because the mock returns what the test expects, not what the real system would
No performance testing	AI generates functional tests. Performance, concurrency and load testing still require human-designed plans and specialist tools

Exam pointer

The second row is the sharpest point in this chapter and generalises beyond AI: *tests generated from code can only ever confirm what the code does, never what it ought to do*. A missing requirement produces no failing test. This is why requirements-based testing (§9.6) and human-derived acceptance criteria remain indispensable.

What the engineer still owns

Define what “correct” means	AI cannot know your business rules, regulatory constraints or users’ expectations. You define the criteria the tests check
Verify expected values	When AI writes <code>assert result == 3.75</code> , confirm that 3.75 is actually correct. AI generates syntactically valid but logically wrong assertions
Identify missing scenarios	Business-level scenarios — “what happens at 09:00 with 5,000 concurrent students?” — require human insight
Review for completeness	Run mutation testing on AI-generated tests. If mutations survive, add assertions until the score is acceptable
Own the test strategy	What to test, how thoroughly, at what stage, with what tools is an engineering decision. AI executes parts of the strategy; you design it
Keep tests up to date	AI generates tests once. When code changes, generated tests may pass incorrectly because a mock still returns old behaviour

9.10 Chapter summary

LO1	Testing has two goals — validation and defect discovery — and forms the dynamic half of V&V alongside static inspection. Tests prove that bugs exist, never that they do not
LO2	Black box tests derive from specifications (system, release, acceptance); white box tests derive from code (unit, integration). Professional teams use both
LO3	Equivalence partitioning groups inputs behaving identically — one test per group. Boundary value analysis tests min−1, min, min+1, max−1, max, max+1
LO4	Unit tests check functions in isolation (fast, isolated, repeatable, automated, self-checking); integration tests check interfaces; system tests check emergent behaviour. Coverage measures thoroughness but not correctness
LO5	TDD: Red (failing test) → Green (minimum code) → Refactor. It yields coverage by construction, early defect detection, better design and a regression suite
LO6	Release testing is done by a <i>separate</i> team: requirements-based tests, scenario tests, and load / stress / spike / endurance testing
LO7	Alpha at the developer's site; beta in the user's environment; acceptance as a formal six-stage gate. In agile, acceptance is continuous at Sprint Review
LO8	AI generates tests quickly, including edge cases humans miss, but cannot invent missing business rules. Engineers verify expected values, add missing scenarios and own the strategy

Further reading for this chapter

Sommerville, *Software Engineering* 10th ed. — Chapter 8, *Software Testing* (the primary reading, and the source of the interface-error classification and testing guidelines used above). Pressman 9th ed. — Chapters 19–22.

Tools referenced: pytest · JUnit 5 · Jest · Postman · REST Assured · Apache JMeter · k6 · Locust · PIT · mutmut · Stryker · Diffblue Cover.

Worked example: one system, four models

This appendix models a single system — the Smart Library Management System from Tutorial 3 — four different ways. Comparing the four diagrams side by side is the fastest way to internalise what each notation is *for*: they describe the same reality, but each one deliberately discards what the others emphasise.

A.1 The scenario

Smart Library Management System (Tutorial 3, Scenario 1)

A well-established urban library faces increasing difficulty managing a growing collection. The manual system produces misplaced records, delays in issuing books, and confusion when tracking overdue items. Management has decided to introduce a digital library management system to handle all daily operations.

Members include **students**, **teachers** and **external visitors**, each with different borrowing privileges. When a new member joins, personal details are recorded and membership is issued after verification by library staff. Members search for books, check availability, and borrow or reserve titles. Some books are in high demand, so reservations must be managed carefully to ensure fairness.

Library staff add new books, update book details, and remove damaged or outdated books. When a member borrows a book, the system records the issue date and calculates a due date. If a book is returned late, a fine is calculated and collected. Members can inquire about their borrowing history and outstanding fines.

The system supports notifications reminding users about due dates or reserved books becoming available. The **librarian** monitors overall activities, generates reports, and ensures smooth operation.

A.2 Step 1 — extract the model elements

Before drawing anything, work through the text systematically.

Actors (roles)	Member (generalised into Student, Teacher, External Visitor) · Library Staff · Librarian · Notification Service (external system)
Use cases (goals)	Search book · Check availability · Borrow book · Return book · Reserve book · View borrowing history · Pay fine · Register member · Verify membership · Add book · Update book · Remove book · Generate reports · Send notification
Candidate classes (nouns)	Member, Student, Teacher, ExternalVisitor, Membership, Book, BookCopy, Loan, Reservation, Fine, Staff, Librarian, Notification, Report
Discarded nouns	“personal details” (attributes of Member) · “due date” (attribute of Loan) · “system” (the thing being built) · “library” (the context, not an object)
Key business rules	Borrowing privileges differ by member type · membership is issued only after staff verification · a fine is calculated when a book is returned late · reservations must be managed fairly

Exam pointer

The distinction between *Book* and *BookCopy* is the single most valuable modelling insight in this scenario. A *Book* is a title — ISBN, author, publisher. A *BookCopy* is a physical item on a shelf with its own barcode and condition. A member borrows a *copy*, not a title, and the library may hold five copies of one title. Modelling both is what makes availability checking, reservations and fine tracking work correctly.

A.3 Step 2 — the use case diagram (external view)

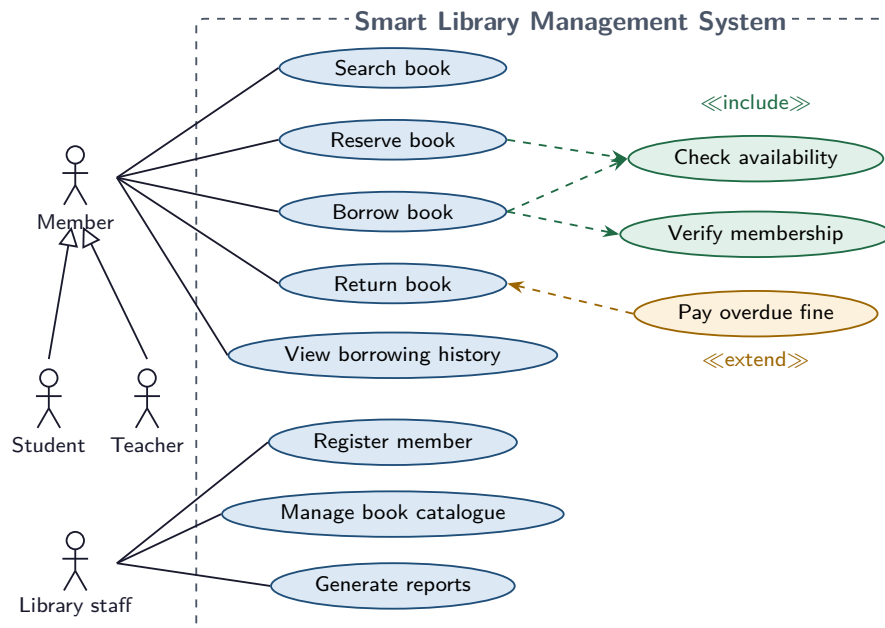


Figure A.1: Use case diagram. Note that Student and Teacher inherit every Member association through the generalisation, so those lines are not repeated.

Key idea

What this diagram tells you: who can do what. What it deliberately does *not* tell you: any sequence, any data, any condition, any timing. That is the point of an abstraction.

A.4 Step 3 — the class diagram (structural view)

The class diagram is split across two figures for legibility: the member hierarchy first, then the core structure it participates in.

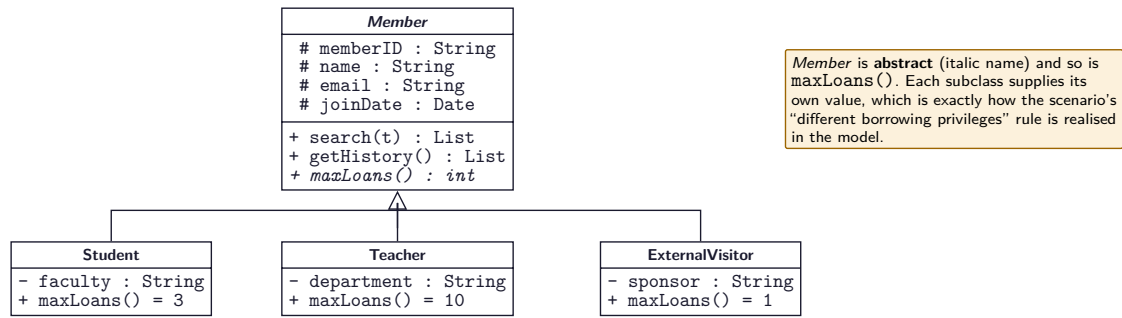


Figure A.2: The member hierarchy. One generalisation arrow serves all three subclasses.

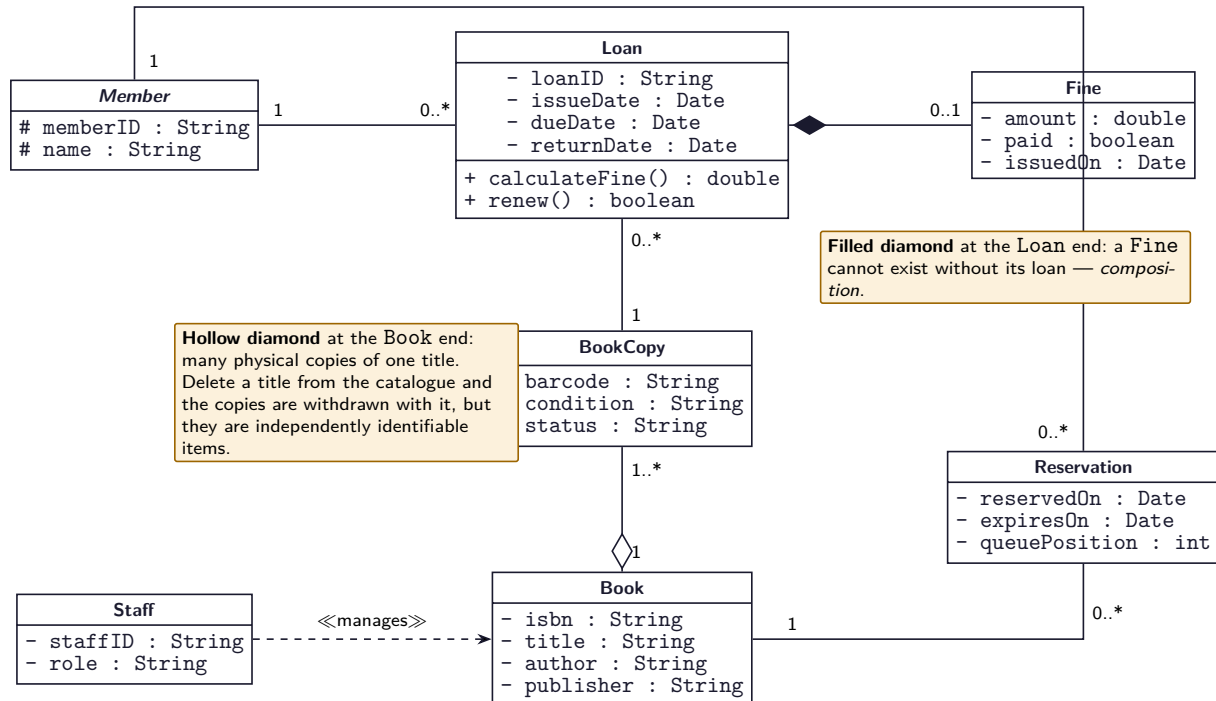


Figure A.3: The core structure. Book is the title; BookCopy is the physical item on the shelf — a member borrows a *copy*.

Scenario sentence	Modelled as	Why
“members include students, teachers and external visitors, each with different borrowing privileges”	Generalisation from abstract Member	The three share identity data but differ in one behaviour; overriding <code>maxLoans()</code> captures exactly that difference
“the system records the issue date and calculates a due date”	Attributes and an operation on Loan	Both belong to the loan event, not to the member or the book
“if a book is returned late, a fine is calculated”	Fine composed into Loan , multiplicity 0..1	A fine cannot exist without its loan; at most one fine per loan
“reservations must be managed carefully to ensure fairness”	<code>queuePosition</code> attribute on Reservation	Fairness is a first-in-first-out rule, which needs an explicit ordering attribute
“library staff add, update and remove books”	Dependency Staff --> Book	Staff operate on books but do not structurally contain them

A.5 Step 4 — the sequence diagram (interaction view)

One scenario only: a member borrows a book, and the copy turns out to be available.

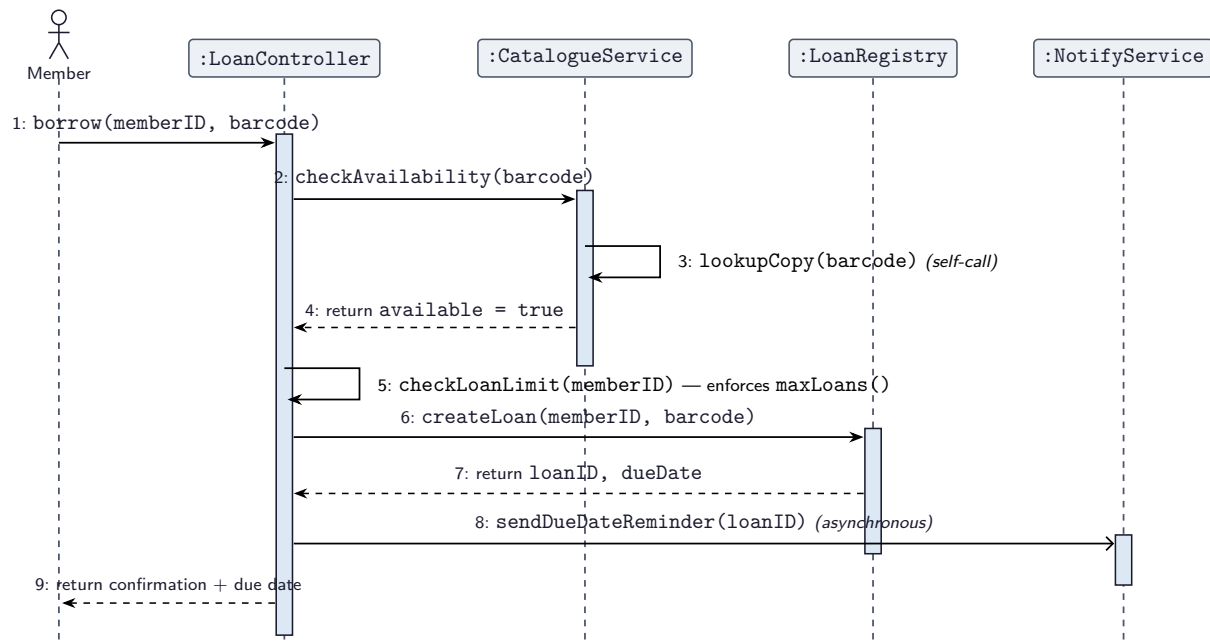


Figure A.4: Sequence diagram: borrowing a book. Message 8 is asynchronous — the controller does not wait for the notification service before confirming to the member.

Key idea

The sequence diagram is the only one of the four that reveals *ordering* — that the loan limit is checked *after* availability but *before* the loan record is created. Neither the class diagram nor the use case diagram can express that, and it is exactly the sort of decision that causes defects when left unstated.

A.6 Step 5 — the activity diagram (behavioural view)

The same borrowing process, but now showing all the branches the sequence diagram's happy path omitted.

A.7 Step 6 — comparing the four models

Diagram	What it shows about the library system	What it hides
Use case	Member, Student, Teacher and Staff and the goals each can pursue; that availability checking is shared behaviour; that fine payment is conditional	Every sequence, condition, data item and rule
Class	That Book and BookCopy are different things; that a Fine belongs to a Loan; that borrowing privileges are a subclass responsibility	Anything that happens over time
Sequence	That the loan limit is checked after availability and before the loan is created; that the reminder is fired asynchronously	Every alternative path except the ones drawn in fragments
Activity	Every refusal branch and every guard in the process	Which object performs each step (unless swimlanes are added)

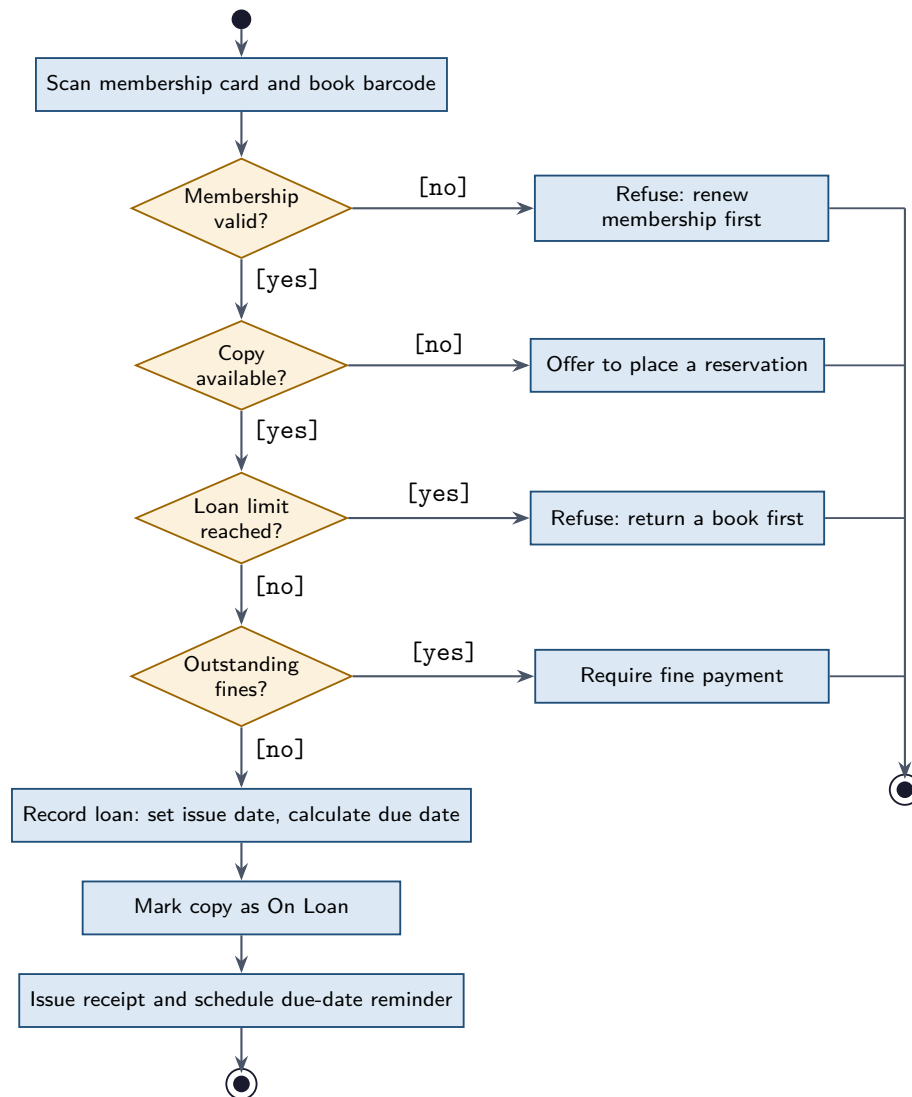


Figure A.5: Activity diagram: the complete borrowing workflow, including every refusal path. Compare the number of branches here with the single path in the sequence diagram.

Exam pointer

When a question says “model this scenario”, the marks are usually distributed across: identifying the correct elements, using the correct notation symbols, getting relationship *directions* right, and supplying multiplicities or guards. Notation errors are cheap to avoid and expensive to make. Work through Chapter 7’s “common errors” tables before any modelling assessment.

Quick reference

B.1 The one-page map of the subject

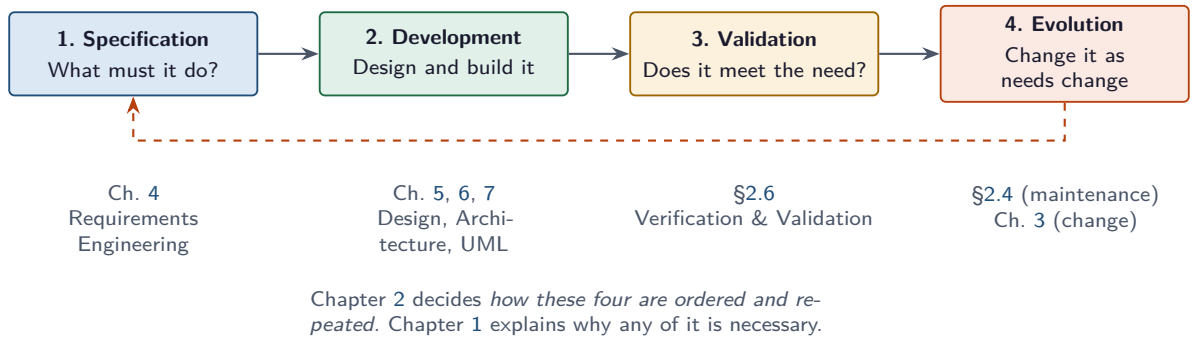


Figure B.1: The four fundamental activities, and where each is covered.

B.2 Definitions worth reproducing exactly

Software	Computer programs, procedures, and associated documentation and data pertaining to the operation of a computer system. (IEEE)
Software engineering	The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software. (IEEE Std 610.12)
Software process	A structured set of activities required to develop a software system, defining what to do, in what order, who is responsible, and how to verify it is done.
Requirement	A condition or capability that must be satisfied — needed by a user to achieve a goal, or required of a system to meet a standard or constraint.
Requirements engineering	The systematic process of discovering, analysing, documenting, validating and managing the requirements for a software system.
Verification	“Are we building the product right?” — checking the software against its specification.
Validation	“Are we building the right product?” — checking the software against the users’ real needs.
Software design	The activity that transforms requirements into a blueprint defining the parts of the system, how they connect, and what each is responsible for.
Software architecture	The fundamental organisation of a system: its major components, the relationships between them, and the principles guiding their design and evolution. (IEEE 1471)
Cohesion	A measure of how closely related the responsibilities within a single module are. High is good.
Coupling	A measure of the degree of interdependence between modules. Low is good.
Abstraction	Representing a system by focusing only on what is relevant for a purpose, suppressing everything else.
Information hiding	Hiding a module’s internal implementation so that other modules see only its interface.
Architectural pattern	A stylised description of good design practice, tried and tested in different environments.
Prototype	An initial version of a system built to demonstrate concepts and try out design options.
Transaction	A coherent sequence of operations that satisfies a goal, and which either completes entirely or has no effect.
Scrum	A lightweight framework helping teams deliver valuable products through short, repeatable cycles called sprints.
Definition of Done	A shared understanding of what “complete” means; a backlog item is not done until it meets every criterion.
Kanban	A method for visualising work, limiting work-in-progress, and managing the flow of value from request to delivery.
WIP limit	A maximum number of work items allowed in a workflow stage at any one time.
Traceability	The ability to follow a requirement’s life in both directions — back to the need that created it, forward to the design, code and tests.
UML	The internationally accepted standard notation for modelling software systems, maintained by the OMG.
Implementation	The process of translating a software design specification into executable source code.
Technical debt	The future cost of additional work created by choosing an easy solution now instead of a better one; it accrues interest.
Code review	A process in which developers examine code written by another, before it is merged into the main codebase.
Version control	A system that records every change made to every file over time, enabling history, collaboration and safe experimentation.
Open source software	Software whose source code is publicly available to read, modify and distribute, subject to its licence.
Prompt engineering	The discipline of crafting instructions to AI systems that produce useful, accurate and well-structured output.
Software testing	Executing a program with the intent of finding defects, and verifying that it does what it is supposed to do.
Equivalence partition	A group of inputs for which the program is expected to behave identically; one test per partition suffices.
Test coverage	The percentage of code actually executed when the test suite runs.
Test-Driven Development	Writing a failing test before the code it tests, then the minimum code to pass it, then refactoring.
Regression testing	Re-running all existing tests after a change, to verify nothing that previously

B.3 Distinctions that are commonly confused

Pair	First	Second
Verification / validation	Against the <i>specification</i>	Against the <i>real need</i>
Incremental / iterative	Staging strategy — adds features	Rework strategy — improves what exists
Generic / bespoke	Built for a market; developer owns the spec	Built for one customer; customer owns the spec
Alpha / beta testing	In the <i>developer's</i> environment, by users	In the <i>user's</i> environment, with real data
Cohesion / coupling	Inside one module; want it <i>high</i>	Between modules; want it <i>low</i>
Aggregation / composition	Hollow diamond; part survives the whole	Filled diamond; part dies with the whole
«include» / «extend»	Always happens; arrow base → included	Conditional; arrow extension → base
Decision / merge node	One in, many out, guarded	Many in, one out, unguarded
Fork / join	One flow becomes many, concurrently	Many flows wait and become one
Sprint Review / Retrospective	Inspects the <i>product</i> , with stakeholders	Inspects the <i>process</i> , within the team
Throwaway / evolutionary prototype	Discarded; reduces requirements risk	Becomes the product; delivers early value
Product Owner / Scrum Master	Owns <i>what</i> and priority	Owns <i>how the team works</i> ; removes impediments
Functional / non-functional requirement	<i>What</i> the system does	<i>How well</i> it does it; must be measurable
SRS / product backlog	Complete, formal, written upfront	Ordered, evolving, detailed just in time
Architecture / design	Structure of the whole system	Decisions inside that structure
Class diagram / sequence diagram	Structural; no clock	Behavioural; strictly time-ordered
Fault avoidance / fault tolerance	Prevent errors entering the code	Handle errors gracefully when they occur
Walkthrough / inspection	Informal; author presents; team traces test cases	Formal; defined roles; checklist-driven; recorder logs defects
Lint / formatter	Finds errors and style problems without running the code	Rewrites code to a consistent style
Git / GitHub	The version-control tool on your machine	The website hosting repositories for collaboration
Permissive / copyleft licence	MIT, Apache, BSD — use freely, attribute	GPL — derivative work must also be open source
Working directory / staging area / repository	Modified	Staged (<code>git add</code>) / committed (<code>git commit</code>)
Black box / white box testing	Designed from the <i>specification</i> ; sees inputs and outputs only	Designed from the <i>code</i> ; sees branches, conditions and loops
Static / dynamic V&V	Inspection — no execution, can start before code exists	Testing — executes the software, catches behavioural errors
Validation / defect discovery	Tests designed to <i>demonstrate</i> correct behaviour	Tests designed to <i>break</i> the system
System / release testing	Finds bugs in the integrated system	Verifies readiness for external use, by an <i>independent</i> team
Line / branch / condition coverage	Every line executed	Every if-true <i>and</i> if-false path / every individual condition both ways
Mutation killed / survived	A test fails when the defect is injected — the suite is strong	All tests still pass — the suite does not check that behaviour
Load / stress testing	At the <i>expected</i> concurrent load	<i>Beyond</i> capacity, to learn how the system breaks

B.4 Numbers worth remembering

60–80%	Share of total lifecycle cost spent on maintenance
50 / 25 / 20 / 5%	Split of maintenance effort: perfective / adaptive / corrective / preventive
~50%	Share of software defects traceable to requirements errors
~80%	Share of project rework caused by poorly understood or changing requirements
100×	Relative cost of fixing a requirements defect after deployment versus at requirements time
60–90%	Share of defects caught by formal inspection before testing (Fagan)
35–45 / 40–50 / 10–20%	Project outcomes: successful / challenged / failed
1968	NATO Software Engineering Conference, Garmisch — the “software crisis”
1970 / 1988 / 1999 / 2001	Waterfall (Royce) / Spiral (Boehm) / XP (Beck) / Agile Manifesto
4 values, 12 principles	The Agile Manifesto
3 roles, 3 artifacts, 5 events	Scrum
7 principles, 7 wastes	Lean software development
6 core practices	Kanban
8 principles	ACM/IEEE-CS Code of Ethics
14 diagram types	UML 2.5.1 (6 required for this course)
25–35 / 15–20 / 25–30%	Development effort: requirements and design / implementation / testing and QA
10×	Relative cost of a defect found after deployment versus in code review
5	SOLID principles; also the layers of a development environment
2005	Git created by Linus Torvalds
72 characters	Recommended maximum length of a commit message summary line
80%+	Line coverage typically required in professional projects; also the target mutation score on critical logic
6 values	Boundary value analysis: min–1, min, min+1, max–1, max, max+1
1969	Dijkstra: “testing can reveal the presence of errors — but not their absence”
5 stages	Unit, integration, system, release, acceptance
6 stages	The acceptance testing process, from defining criteria to formal accept or reject

B.5 Answer templates for common question types

Question type	Structure that earns full marks
“Recommend a process model for... and justify”	Name the model. Then map three axes to specific model properties: (1) requirements stability, (2) risk and criticality, (3) customer availability and contract type. Finish with one honest limitation of your choice.
“Compare X and Y”	Use a table with named comparison dimensions down the left. Never write two paragraphs of prose — the marks are per dimension, and a table makes each one visible.
“Write requirements for...”	Give one functional, one non-functional (<i>quantified</i>) and one domain requirement. Use “The system shall...” phrasing. Then state how each would be tested.
“Draw a diagram for this scenario”	Extract elements first (actors, use cases, classes, steps). Draw. Then label every relationship direction and multiplicity or guard. State one assumption you made about anything ambiguous in the scenario.
“Explain why X matters”	Define X. Give the mechanism by which it produces its benefit. Give one concrete example. Give one consequence of its absence.
“Design test cases for...”	Name the technique before applying it. Equivalence partitioning: state the partitions, then one representative test each. Boundary analysis: give all six boundary values. Present them in the test-case table format (ID, description, preconditions, steps, expected result) — the structure itself earns marks.
“Recommend a language / justify”	Name it, then balance <i>technical</i> fit (domain, performance, safety, platform, libraries) against <i>non-technical</i> constraints (team expertise, timescales, maintenance, organisational standards). Concede the constraint that decided it.
“Evaluate the role of AI in...”	Two columns: what AI does well, what remains human. Close with accountability — the engineer who accepts AI output owns it.

Tutorial coverage map

Tutorial	Topic	What it drills	Covered in
1	Flowcharts and development approaches	Ten scenarios (grade analyser, fare management, shop billing, payroll, library fines, tiered electricity billing, banking simulator, event registration, restaurant orders, parking fees), each naming a development approach — modular, incremental, iterative, top-down, prototype, stepwise refinement, user-centred	§7.2; Ch. 2 for the approaches
3	Use case diagrams	Four scenarios: Smart Library, Online Food Ordering, Hospital Appointment, Vehicle Rental. Identify actors and interactions, then draw	§7.3; Appendix A
4	Class diagrams	Four scenarios: Property Rental, Pharmacy Inventory, Multi-Vendor E-Commerce, Conference Paper Review. Identify classes, attributes, operations, visibility, and structural relationships with accurate multiplicity	§7.4; Appendix A
5	Jira	Setting up a project, understanding backlog / board / roadmap / reports, creating epics, stories, tasks and bugs, linking issues, backlog management	§3.9
6	Trello	Workspace and board setup, six-list structure, card details, moving cards, automation rules, activity tracking	§3.9
7	Activity diagrams	Four library scenarios: applying for a card, reserving a study room, interlibrary loan (with decision, fork and join), workshop registration (with fork and join)	§7.6
8	Sequence diagrams	Lifelines, messages, activation, creation and destruction, combined fragments (alt , opt , loop), worked login and ATM examples, PlantUML	§7.5; Appendix A
9	Git and GitHub	Set up Git; create a repository; make three commits on a branch; open a pull request; try an AI assistant on a small function	§8.6, §8.5, §8.8
10	Software testing	Design test cases by equivalence partitioning and boundary value analysis; write unit tests in pytest or JUnit; practise the Red–Green–Refactor cycle; measure coverage	§9.3, §9.4, §9.5

C.1 Scenario checklists

The tutorial scenarios recur across assessments. These checklists convert the prose into model elements quickly.

C.1.1 Any use case scenario

1. Who is mentioned by *role*? Those are actors. Merge synonyms; generalise if one role is a kind of another.
2. Which external systems are mentioned (payment, email, SMS, a partner organisation)? Those are actors too.
3. What *goals* does each role have? Each goal is one use case, named as a verb phrase.
4. Which behaviour appears inside several use cases? Factor it out as `<<include>>`.
5. Which behaviour happens only under a stated condition (“if returned late...”) ? That is `<<extend>>`.
6. Draw the boundary last, and check that no actor ended up inside it.

C.1.2 Any class diagram scenario

1. Underline every noun. Strike out synonyms, attributes and things outside the boundary.
2. Look for the phrase “two types of” or “different kinds of” — that is always a generalisation.
3. Look for “consists of” and “contains” — candidate composition. Test it: delete the whole; must the part go too?
4. Look for anything with its own identity and lifecycle that links two other classes — that is an association class or a full class in its own right (Loan, Enrolment, Booking, Prescription line).
5. Distinguish *catalogue entries* from *physical instances* (Book vs. BookCopy, Product vs. StockBatch, Course vs. CourseOffering).
6. Put every operation where its data lives.
7. Add multiplicity at both ends of every association before you stop.

C.1.3 Any activity diagram scenario

1. Number the steps in the prose. Each numbered step is usually one action node.
2. Every “if”, “check whether” or “is it valid?” becomes a decision node with labelled guards.
3. Every “incorrect submissions are returned for correction” becomes an edge back to a merge point.
4. Every “and” where two things happen independently becomes a fork; the point where both must finish becomes a join.
5. If the prose names who performs each step, use swimlanes.
6. Confirm exactly one initial node and that every path reaches a final node.

C.1.4 Any test design exercise

1. Decide **black box** or **white box** first — it determines everything that follows.
2. Identify the **valid range** from the specification, then write down the partitions: one invalid-low, one or more valid, one invalid-high.
3. Take **one representative value** from each partition. Testing more values from the same partition earns nothing.
4. Add the **six boundary values**: min−1, min, min+1, max−1, max, max+1.
5. Add cases for every **error condition** named in the specification — each exception the function can raise needs its own test.
6. Present the result as a **table** with ID, description, preconditions, steps and expected result. The structure carries marks.
7. State the **expected** result for every case before running anything; a test whose expected value is decided after seeing the output verifies nothing.

C.1.5 Any Git or collaboration exercise

1. `git pull origin main` first — never start work on a stale copy.
2. Create a branch named for the work: `feature/`, `fix/` or `docs/` plus a short description.
3. Commit in small, coherent steps. Each message: a summary under 72 characters saying *what*, then *why* if it is not obvious.
4. `git status` before every `add` and `commit` — confirm you are staging what you think you are.
5. Push the branch, open a pull request, and let CI run before requesting a human reviewer.
6. Address review comments by pushing further commits to the same branch — do not open a new PR.
7. After approval, merge and delete the branch.
8. If a merge conflict appears: open the file, remove the «««<, ===== and »»»> markers, keep or combine the correct code, then `git add` and `git commit`.

C.1.6 Any sequence diagram scenario

1. Pick *one* scenario. A sequence diagram shows one path, not the whole use case.
2. List the participants: the initiating actor, then one object per responsibility (controller, service, registry, database, external gateway).
3. Order the messages strictly. Ask at each step: who must know the answer before the next step can happen?
4. Use synchronous arrows where the caller waits, asynchronous where it does not (notifications, logging, email).
5. Draw a return for every synchronous call that produces a value.
6. Wrap the branch in an `alt` fragment only if the branch is essential to the story; otherwise draw the happy path and note the alternative in prose.
7. Check every activation bar covers exactly the period the object is working.

Glossary

Abstraction

Focusing only on what is relevant for a particular purpose and suppressing everything else.

Acceptance criteria

Specific, verifiable conditions that must be true for a user story to be considered complete.

Acceptance testing

Testing by the customer with real data, in conditions representative of real use; the formal gate before delivery.

ADR (Architecture Decision Record)

A short document recording an architectural decision: its context, the decision taken, and its consequences.

Aggregation

A weak whole–part relationship in which the part can exist independently of the whole. Hollow diamond at the whole end.

Agile

A development culture that embraces change, delivers working software frequently in short cycles, and favours collaboration over contract negotiation.

Alpha testing

Testing carried out by users in the developer’s environment.

Architectural pattern

A stylised, reusable description of good architectural practice, tested across different environments.

Association class

A class attached to an association, used when the relationship itself carries data.

Backlog

An ordered, evolving list of everything a product might need. The agile alternative to a requirements specification.

Branch

A separate line of development in Git; created to work on a feature or fix, then merged back into the main branch.

Assertion

A statement inside a test declaring what must be true; a test without meaningful assertions can execute code without checking anything.

Baseline

An agreed, approved set of requirements or artefacts against which change is controlled.

Black box testing

Test design based on the *specification*: only inputs and outputs are visible. Used for system, release and acceptance testing.

Boundary value analysis

A test design technique that tests the edges of valid input ranges — $\text{min}-1$, min , $\text{min}+1$, $\text{max}-1$, max , $\text{max}+1$ — where defects cluster.

Beta testing

Testing carried out in the users' own environment with real data before general release.

Bespoke software

Software built for a single named customer to that customer's specification.

Burndown chart

A chart of remaining work against time within a sprint.

Commit

A saved snapshot of a Git project at one point in time, recording what changed, who changed it, when, and a message explaining why.

Copyleft

A licence condition requiring derivative works to carry the same open-source licence. GPL is the standard example; sometimes called "viral".

Condition coverage

The proportion of individual conditions within decisions that have been tested as both true and false. The strongest of the three common coverage measures.

CBSE

Component-Based Software Engineering — assembling systems from existing reusable components.

CI/CD

Continuous Integration and Continuous Delivery/Deployment — automating build, test and release so that changes reach production quickly and safely.

Client-server

An architectural pattern where servers provide services and clients consume them across a network.

CMMI

Capability Maturity Model Integration — a five-level model of organisational process maturity.

Cohesion

How closely related the responsibilities within a single module are. High cohesion is desirable.

Combined fragment

A framed region of a sequence diagram expressing alternation (`alt`), option (`opt`), repetition (`loop`) or concurrency (`par`).

Composition

A strong whole-part relationship with coincident lifetimes: the part cannot exist without the whole. Filled diamond at the whole end.

Context model

A model defining the system boundary and every external entity the system interacts with.

Corrective maintenance

Fixing defects found in the live system. Roughly 20% of maintenance effort.

Coupling

The degree of interdependence between modules. Low coupling is desirable.

Dependency

A library or package that your code relies on; managed by a package manager such as pip, npm or Maven.

Defect discovery

The testing goal of designing test cases to *break* the system and reveal hidden defects, as opposed to demonstrating correct behaviour.

Cycle time

The elapsed time from starting a work item to finishing it. The primary Kanban flow metric.

Data flow diagram

A model of how data moves through a system and is transformed, using external entities, processes, data stores and flows.

Definition of Done

A shared, explicit standard that a backlog item must meet to be considered complete.

Dependability

The combination of reliability, availability, safety and security that makes a system trustworthy.

DevOps

A culture and set of practices uniting development and operations so that changes reach production quickly, safely and repeatably.

Domain requirement

A constraint arising from the industry, legal regulation or operating environment.

Docstring

Structured documentation attached to a function, recording its purpose, parameters, return value and errors.

Emergent behaviour

System properties that appear only when the whole system runs together, and which therefore cannot be found by testing components individually.

Endurance testing

Running a system under normal load for an extended period, to reveal memory leaks and resource exhaustion.

Equivalence partitioning

A test design technique that divides inputs into groups the program should treat identically, testing one representative value per group.

Elicitation

The active discovery and co-creation of requirements with stakeholders — distinct from mere “gathering”.

Epic

A large body of work in an agile backlog, too big to complete in one sprint, which decomposes into features and stories.

Evolutionary prototype

A prototype that is progressively refined until it becomes the delivered system.

Feasibility study

A short early assessment of whether a proposed system is technically, economically, legally, operationally and schedule-wise viable.

Fault avoidance

Writing code that contains no errors in the first place — standards, strong typing, small functions, tests, review.

Fault tolerance

Writing code that handles errors gracefully when they occur — exception handling, defensive programming, logging, graceful degradation.

Git

The distributed version-control system created by Linus Torvalds in 2005; runs on the developer's own machine.

GitHub

A web platform hosting Git repositories, adding pull requests, issues, Actions (CI/CD) and Copilot. GitLab and Bitbucket are alternatives.

FIRST properties

The qualities of a good unit test: Fast, Isolated, Repeatable, Automated (and) Self-checking.

Formal inspection

A structured, checklist-driven examination of a document or code by a team, with a recorder logging every defect.

Functional requirement

A statement of what the system must do — its behaviours, services and features.

Generalisation

The “is-a” inheritance relationship. Hollow triangle pointing at the superclass.

Generic software

A product developed for a broad market, with the developer controlling the specification.

Hallucination

An AI system confidently generating something that does not exist — for code, typically a plausible-sounding library function that is not real.

IDE

Integrated Development Environment — a workspace that understands the language, offering autocompletion, error detection, navigation and debugging.

HLD / LLD

High-Level Design (architecture and subsystems) and Low-Level Design (component internals, interfaces, algorithms).

Incremental development

Building and delivering a system in successive functional slices, each adding usable capability.

Information hiding

Concealing a module's internal implementation so that other modules depend only on its interface.

INVEST

A checklist for user story quality: Independent, Negotiable, Valuable, Estimable, Small, Testable.

Iterative development

Repeatedly revisiting and improving parts of a system that have already been built.

Lint

A tool that analyses code for errors, bugs and style problems without running it — ESLint, Pylint, Checkstyle, SonarQube.

Interface error

A defect at the boundary between components, classified as interface misuse, interface misunderstanding, or a timing error. Invisible to unit testing.

Line coverage

The proportion of source lines executed by at least one test. The most basic coverage measure; 80%+ is typical in professional projects.

Load testing

Testing under the *expected* number of concurrent users, as opposed to stress testing which exceeds capacity deliberately.

Kanban

A method for visualising work, limiting work-in-progress and managing flow, without prescribed sprints or roles.

Layered architecture

Organising a system into horizontal layers, each providing services to the layer above and using the layer below.

Lean

A philosophy, adapted from the Toyota Production System, of eliminating waste and optimising the whole value stream.

Lifeline

The dashed vertical line in a sequence diagram representing a participant's existence over time.

Maintainability

The ease with which a system can be understood, corrected, adapted and enhanced.

Merge conflict

The situation where two developers change the same part of the same file, so Git cannot merge automatically and marks both versions for manual resolution.

Mock

A test double that stands in for a real dependency such as a database or network service, so a unit test remains isolated and fast.

Mutation testing

Introducing deliberate small defects into source code to check whether the test suite detects them. A surviving mutation indicates a weak test.

Microservices

An architectural style in which each capability runs as a small, independently deployable service with its own data store.

MoSCoW

A prioritisation technique: Must have, Should have, Could have, Won't have (this release).

Multiplicity

The number of instances that may participate at one end of an association, e.g. 1, 0..1, 1..*, *.

MVC

Model-View-Controller — an architectural pattern separating data and rules, presentation, and user-interaction handling.

Non-functional requirement

A statement of how well the system must perform — a quality attribute, which must be measurable.

Package manager

A tool that downloads, installs and updates a project's dependencies, recording versions in a file — pip, npm, Maven, Gradle.

Path testing

A white-box strategy that aims to exercise every independent execution path through a component, so every statement runs and every condition is tested both ways.

Pair programming

Two developers working at one machine, one writing (driver) and one reviewing and planning ahead (navigator).

Perfective maintenance

Improving performance, usability or functionality in response to user feedback. Roughly 50% of maintenance effort.

Pipe and filter

An architectural pattern in which discrete transformation components pass data along in sequence.

Plan-driven

A development culture that plans comprehensively upfront and controls change through formal approval.

Product Owner

The Scrum accountability for what is built and in what order; owns and prioritises the Product Backlog.

Prompt engineering

The discipline of crafting instructions to an AI system so that it produces useful, accurate and well-structured output.

Pull request

A formal request to merge one branch into another, reviewed and commented on before merging. The standard modern code-review workflow.

Postman

The industry-standard GUI tool for designing, sending and testing API requests.

Prototype

An initial version of a system built to demonstrate concepts and try out design options.

RAD

Rapid Application Development — an incremental model using parallel component construction on a very short cycle.

Refactoring

Improving the internal structure of code without changing its external behaviour.

Regression testing

Re-running existing tests after a change, to confirm that previously working behaviour still works.

Release testing

Testing a complete version intended for use outside the development team, carried out by a separate, independent test team.

Repository (Git)

The complete history of a project — every file and every change — stored in a hidden `.git` folder.

Repository pattern

An architecture in which all components communicate only through a shared central data store.

Requirements engineering

The systematic process of discovering, analysing, documenting, validating and managing requirements.

Retrospective

The Scrum event in which the team inspects and improves *its own process*.

Reuse-oriented development

Assembling a system largely from existing components, negotiating requirements against what is available.

Scenario testing

Release testing driven by realistic end-to-end stories of system use, rather than by individual requirements.

Scrum

A lightweight framework for delivering products through short, timeboxed sprints, with three roles, three artifacts and five events.

Scrum Master

The Scrum accountability for how the team works: facilitates events, removes impediments, coaches the team. Not a project manager.

Separation of concerns

Structuring a system so that each part deals with one concern and is insulated from changes to others.

Software crisis

The mid-twentieth-century recognition that software projects routinely ran over budget, late, and delivered the wrong system — formally raised at the 1968 NATO conference.

SOLID

Five principles for maintainable object-oriented code: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion.

Staging area

The intermediate Git state, between the working directory and the repository, holding changes marked for the next commit.

Spike testing

Simulating a sudden peak in load, such as all users arriving the moment a service opens.

Spike

A short, timeboxed investigation or prototype used to answer a technical question before committing to a design.

Spiral model

A risk-driven evolutionary process in which each loop of the spiral is one phase, beginning and ending with risk assessment.

Sprint

A timeboxed iteration of one to four weeks that produces a potentially shippable increment.

SRS

Software Requirements Specification — the formal, complete description of what a system must do and how well.

Stakeholder

Any person, group or organisation with an interest in the system or affected by it.

State machine diagram

A model of the states an object can occupy and the events that cause transitions between them.

Static analysis

Automated examination of source code without executing it, checking for defect patterns and standard violations.

Swimlane

A partition in an activity diagram attributing each action to a responsible actor or organisational unit.

TDD

Test-Driven Development — write a failing test, write the minimum code to pass it, then refactor.

System testing

Testing the complete integrated system against its requirements; its distinctive target is emergent behaviour.

Test case

A complete specification of one test: identifier, description, preconditions, steps, expected result and actual result.

Test coverage

The percentage of code executed when the test suite runs. Measures what was executed, not whether it was checked correctly.

Technical debt

The future cost incurred by choosing an expedient solution now instead of a better one.

Test double

Any stand-in for a real dependency in a test — mocks, stubs and fakes.

Throughput

The number of work items completed per unit time. A Kanban flow metric.

Throwaway prototype

A prototype built quickly to resolve requirements ambiguity, then discarded before the real system is built.

Traceability

The ability to follow a requirement backward to its originating need and forward to its design, code and tests.

Unit

The smallest independently testable piece of software — a single function, method or class.

Unified Process

A hybrid, use-case-driven, architecture-centric, iterative process framework with four phases: Inception, Elaboration, Construction, Transition.

UML

Unified Modeling Language — the standard notation for modelling software systems.

Use case

An interaction between an actor and the system that achieves a goal of value to that actor.

User story

A short statement of a requirement in the form “As a [role], I want to [action], so that [benefit]”.

V-model

A process model pairing each design level with a corresponding test level, so test planning begins early.

Validation

Checking that the software meets the users' real needs — “are we building the right product?”

Validation (testing goal)

Designing test cases to demonstrate that the system meets its requirements when used as expected.

Velocity

The number of story points a Scrum team completes per sprint; used to forecast capacity.

Verification

Checking that the software conforms to its specification — “are we building the product right?”

White box testing

Test design based on the *code structure*: branches, conditions and loops are visible. Used for unit and integration testing.

Walkthrough

An informal review in which the author presents work to a group who ask questions and identify issues.

Waterfall

A sequential process model in which each phase is completed and approved before the next begins.

WIP limit

A cap on the number of work items permitted in a workflow stage at any one time.

XP

Extreme Programming — an agile methodology emphasising engineering excellence through TDD, pairing, refactoring, continuous integration and small releases.

YAGNI

“You Ain’t Gonna Need It” — the XP principle of not building speculative functionality.