COMP433

Software Engineering

Software processes and models (part 1)

Hisham Ihshaish

Birzeit University

Semester 2 (Spring) 2025/2026 | Sections 5 & 6

Chapter 2.

What we will cover today

01 **What is a software process?**
Defining processes and their core activities

02 **Plan-driven and agile approaches**
Two philosophies for organising software work

03 **The waterfall model**
Origins, stages, strengths, and limitations

04 **The V-model**
Extending waterfall with explicit test planning

05 **Incremental development**
Building software in slices

What is a software process?

Organising the work of building software

Defining a software process

Sommerville's definition

*A software process is a structured set of activities required to develop a software system. These activities **include** specification, design and implementation, validation, and evolution.*

In plain terms: a software process describes **what you do**, **in what order**, and **with what outputs**, when you build software. It turns an ad hoc collection of programming tasks into a managed, repeatable engineering activity.

A process is not a rigid prescription.
Different projects require different processes. A safety-critical embedded system and a consumer mobile app will follow very different processes, even though both involve the same fundamental activities. The choice of process model is an engineering decision in itself.

Four fundamental process activities

Sommerville identifies four activities that appear in every software process, regardless of the model used.

1

Software specification

Defining what the system should do and the constraints on its operation. This is where requirements are gathered, analysed, and documented.

2

Software design and implementation

Designing the system structure and converting the design into an executable program. Includes architectural decisions and coding.

3

Software validation

Checking that the software meets the customer's requirements. Includes testing, reviews, and acceptance procedures.

4

Software evolution

Modifying the software to adapt to changing customer and market requirements. Most software is never truly finished.

Core and supporting process activities

Core activities

- Specification -- defining what the system should do
- Design and implementation -- building the system
- Validation -- checking it meets requirements
- Evolution -- adapting to change over time

Supporting activities

Configuration management
Tracking changes to software artefacts and ensuring consistency across versions

Quality management
Establishing quality standards and ensuring the process and product meet them

Project management
Planning, scheduling, estimating, and controlling the work

Documentation
Recording decisions, designs, and user guides throughout the lifecycle

Why process matters

Predictability
A defined process lets you estimate timelines, costs, and resource needs. Without it, every project is a gamble.

Quality
Processes embed quality checks at each stage, catching defects early rather than discovering them in production.

Coordination
When multiple people work on a system, a shared process ensures everyone knows what to do, when, and how their work connects to others.

Learning from experience
A defined process can be measured and improved. If you do not know what you did, you cannot improve on it next time.

Plan-driven and agile approaches

Two philosophies for organising software work

Plan-driven versus agile

Plan-driven

- All activities planned in advance before development begins
- Progress measured against the plan
- Detailed documentation at each stage
- Change is managed through a formal process
- Suited to stable requirements and contractual environments

Agile

- Planning is incremental; plans evolve as understanding grows
- Working software is the primary measure of progress
- Documentation is lighter, focused on essentials
- Change is embraced and expected throughout the project
- Suited to evolving requirements and fast-feedback environments

Most real-world projects fall somewhere on a spectrum between these two extremes.

There is no universal process

The choice of process model depends on several contextual factors:

- Project size and complexity
- Requirements stability (are they well-understood or likely to change?)
- Team size, experience, and distribution
- Domain criticality (lives at stake vs. a personal app)
- Regulatory and contractual constraints

Consider two projects

A pacemaker firmware update. Requirements are fixed by medical regulations. Failure can kill a patient. Exhaustive documentation and verification are mandatory. A plan-driven, sequential process with formal reviews makes sense.

A university event-booking mobile app. Requirements are unclear and likely to change after initial user feedback. The team is small and co-located. An incremental or agile approach with frequent releases makes more sense.

The waterfall model

The original sequential process model

Origins of the waterfall model

Historical context

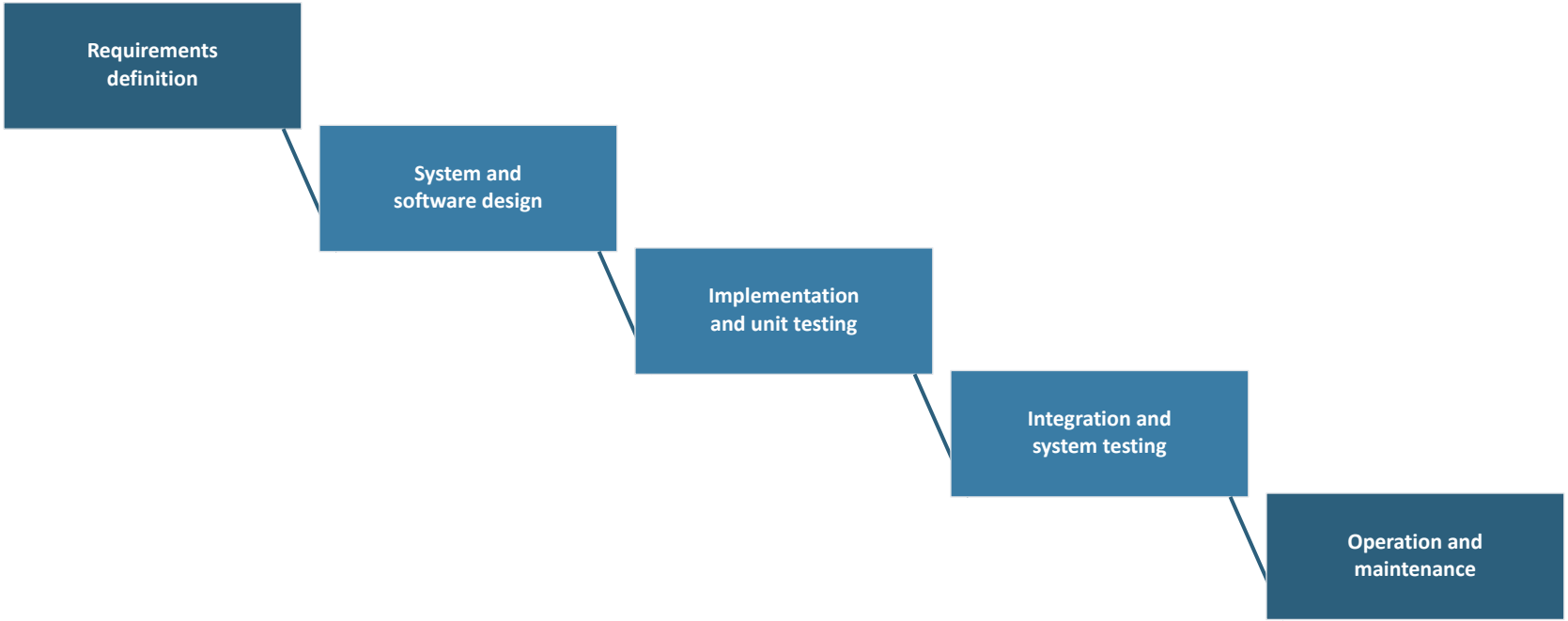
The waterfall model is often attributed to Winston Royce's 1970 paper, "Managing the Development of Large Software Systems". However, Royce never used the word "waterfall", and his paper actually argued against a purely sequential approach.

Royce presented the sequential model as a starting point and then identified its risks. He proposed adding feedback loops between stages and building the system twice (a prototype first, then the production version). The simplified, feedback-free version was adopted by industry and government contracting and became known as the "waterfall".

A common misconception

Many textbooks describe the waterfall as Royce's recommended approach. In fact, Royce's paper included a diagram of the sequential model with the caption that it was "risky and invites failure". He spent the rest of the paper proposing improvements. Understanding this nuance helps us appreciate both the model's value and its limitations.

Waterfall model: stages



Each stage produces documented outputs that feed into the next. In the pure waterfall, a stage is completed before the next begins.

Strengths and limitations of the waterfall

Strengths

- Clear, well-defined stages with documented outputs
- Easy to understand and manage progress against milestones
- Works well when requirements are stable and well understood
- Suitable for contractual and regulated environments that demand sign-off at each stage

Limitations

- Assumes requirements can be fully specified before design begins
- Late discovery of problems: working software only appears at the end
- Inflexible to change once a stage is completed
- Testing comes late; defects found in integration are expensive to fix
- Customers see the product only at delivery, not during development

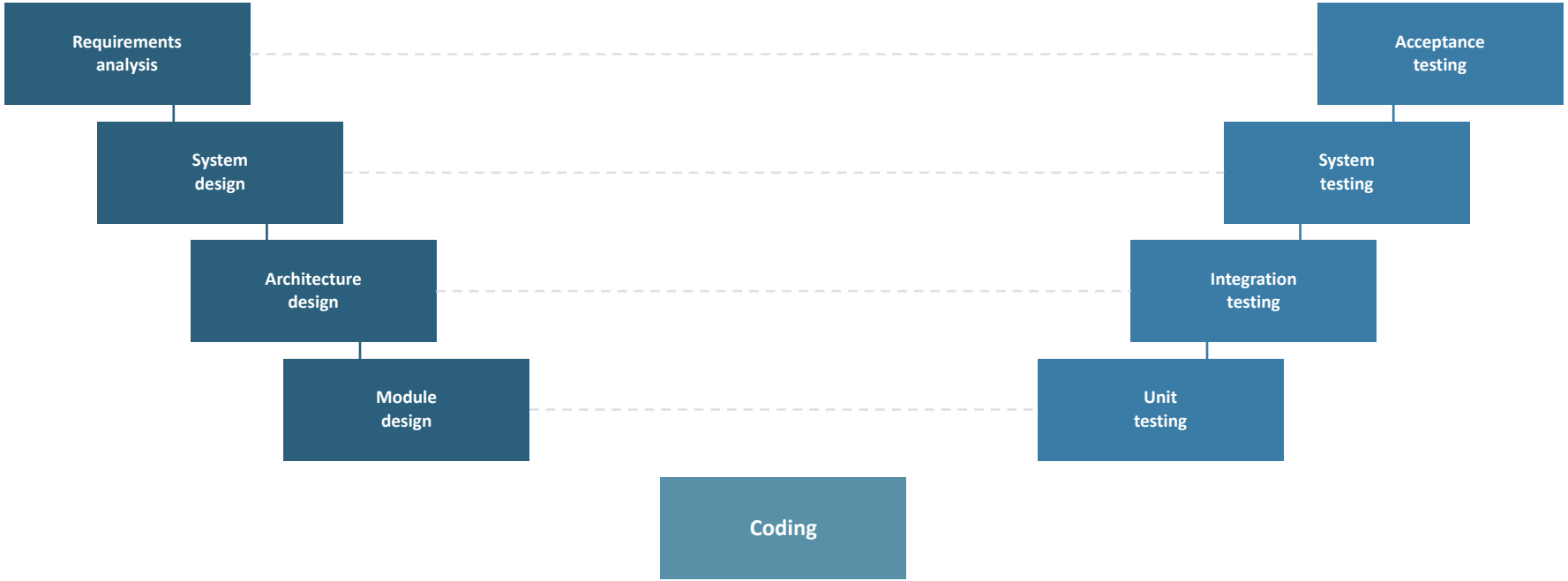
When the waterfall works (and when it does not)

Well-suited	Poorly suited	Common adaptation
Safety-critical embedded systems with stable, well-understood requirements and mandatory regulatory sign-off at each stage.	Web and mobile products where user needs are uncertain and rapid iteration is needed. Waiting months before delivering working software is a competitive disadvantage.	Waterfall with feedback loops between adjacent stages, as Royce originally proposed. This allows limited iteration while preserving the overall sequential structure.
Example: medical device firmware, avionics control systems.	*Example: a new social networking app or e-commerce platform.*	*This hybrid is widely used in practice, particularly for large enterprise systems.*

The V-model

Extending the waterfall with explicit test planning

The V-model



Each development phase (left) has a corresponding test phase (right). Test cases are planned during development, not after.

Waterfall versus V-model

Aspect	Waterfall	V-model
Testing	Single testing phase after implementation	Test phases mirror each development phase
Test planning	Typically done during or after the test phase	Planned in parallel with each development phase
Defect detection	Defects often found late, during integration	Defects can be caught earlier via phase-specific tests
Sequential?	Yes	Yes (still sequential overall)
Flexibility	Low	Low (same rigidity for requirements changes)

The V-model improves on the waterfall's treatment of testing, but does not solve the fundamental problem of handling changing requirements.

Incremental development

Building software in slices

Incremental development: the core idea

> *"An approach where specification, development, and validation are interleaved rather than separate, producing a series of versions (increments), each adding functionality to the previous one."*
>
> *-- Sommerville (2016), Software Engineering, 10th ed., Ch 2*

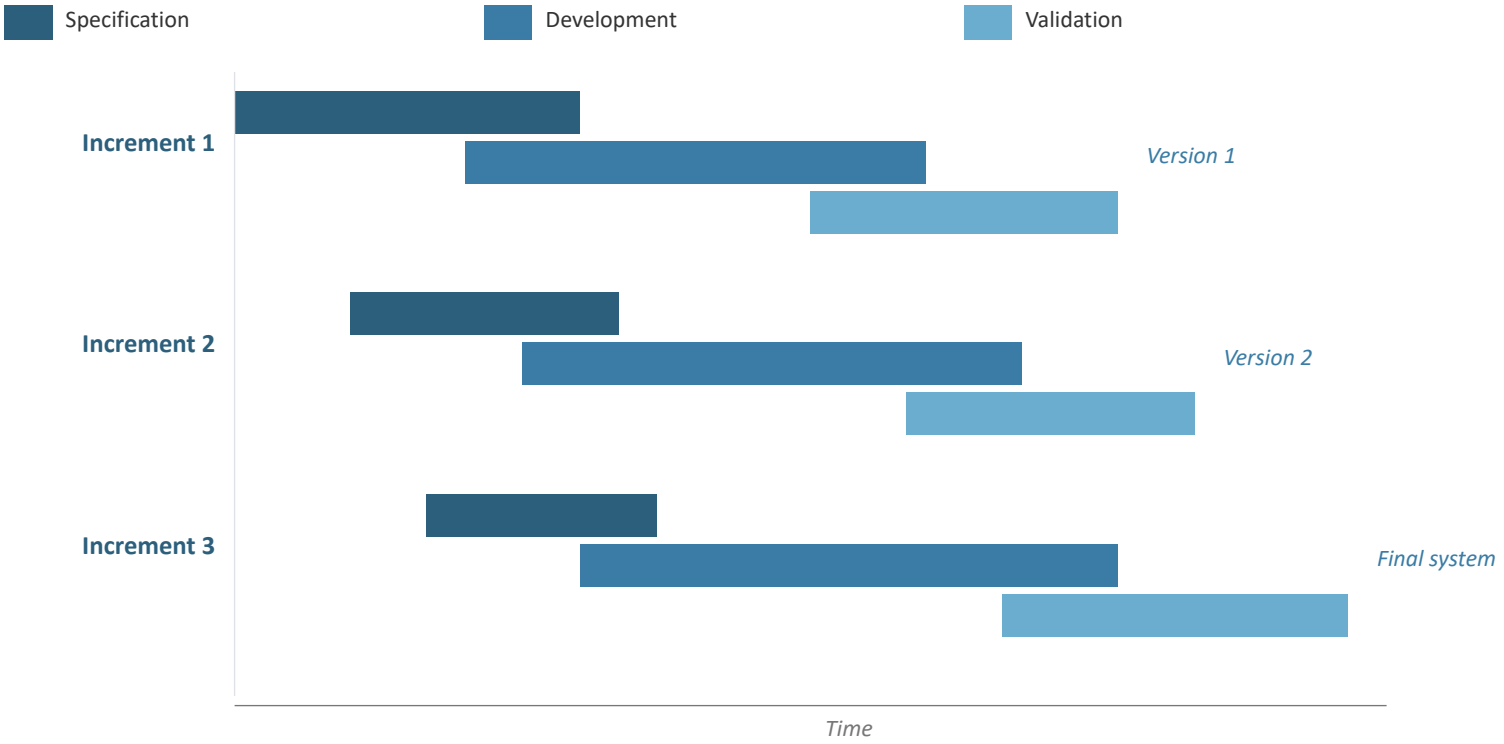
In plain terms: rather than finishing all requirements, then all design, then all coding, then all testing, you develop the system in working slices. Each slice is a usable piece of software that you can show to stakeholders, test with real users, and learn from before building the next one.

Example: a university course registration system: say, Ritaj.

Increment 1	Student login and profile management
Increment 2	Course catalogue browsing and search
Increment 3	Course enrolment and schedule building
Increment 4	Waitlists, prerequisite checks, and advisor approval

How incremental development works

The process activities run concurrently across increments. Each increment produces a deliverable version of the system.



Strengths and challenges

Strengths

- Reduced cost of accommodating changing requirements, because less rework is needed when requirements change between increments.
- Early delivery of value to the customer. The most critical functionality is delivered first, so useful software is available sooner.
- Easier to get meaningful customer feedback on a working system than on abstract specification documents.
- Lower project risk. Even if the project is cancelled, the customer retains the completed increments.

Challenges

- System structure tends to degrade as new increments are added. Without deliberate refactoring, the design can become messy over time.
- Difficult to identify common facilities needed by all increments upfront, which may lead to rework when later increments require shared infrastructure.
- Process visibility is harder for management. Producing documents for every increment is not cost-effective, yet managers need progress markers.
- May not suit large, complex systems with many interdependent subsystems (e.g. embedded systems or safety-critical software).

Plan-driven or agile?

Incremental development is a delivery strategy, not a philosophy. It can be plan-driven (all increments pre-planned with fixed scope) or agile (increment scope determined adaptively through ongoing prioritisation). Most modern projects use some form of incremental delivery, regardless of whether the overall approach is plan-driven or agile.

Three models compared

Waterfall	V-model	Incremental
Flow Strictly sequential; each phase completes before the next begins	**Flow** Sequential with explicit test-planning mirroring each development phase	**Flow** Interleaved; specification, development, and validation overlap across increments
Testing Concentrated at the end, after implementation	**Testing** Planned alongside each phase; executed in reverse order after coding	**Testing** Each increment is validated before the next begins
Flexibility Very low; changes are costly once a phase is signed off	**Flexibility** Low; same rigidity as waterfall, but with stronger quality assurance	**Flexibility** Moderate to high; later increments can adapt based on earlier feedback
First delivery Only at the very end of the project	**First delivery** Only at the very end, after all verification and validation stages	**First delivery** After the first increment, well before the full system is complete
Best suited for Well-understood, stable requirements (e.g. safety-critical, regulatory)	**Best suited for** High-assurance systems where traceability from requirements to tests is critical	**Best suited for** Most business systems, web/mobile apps, projects where early feedback matters
Spectrum Plan-driven	**Spectrum** Plan-driven	**Spectrum** Can be plan-driven or agile

Discussion: choosing a process

Consider three projects. For each, decide which process model (waterfall, V-model, or incremental) you would recommend and why. Type your answers in the Teams chat.

A **Air traffic control system**

Strict safety regulations. Requirements are mandated by aviation authorities. Certification requires complete documentation and formal sign-off at each stage.

B **University event-booking app**

A student team is building a mobile app for booking campus events. Requirements are vague; they plan to learn from early users. Deadline is six months.

C **E-government tax portal**

Fixed regulatory requirements that are well understood, but the system must integrate with multiple existing government databases. Large development team across two sites.

Take 3 minutes. Share your reasoning, not just the model name.

Summary

A software process is a structured set of activities for developing software. It turns ad hoc coding into managed engineering.

Four fundamental activities appear in every process: specification, development, validation, and evolution.

Plan-driven and agile represent two ends of a spectrum. Most projects fall somewhere in between.

The waterfall model is sequential and documentation-heavy. It works for stable requirements but struggles with change. Royce's original proposal included feedback loops.

The V-model extends the waterfall by pairing each development phase with a corresponding test phase, promoting early test planning.

Incremental development builds software in usable slices, enabling early delivery, fast feedback, and easier accommodation of change.

Next session

Software processes and models (part 2)

- The Agile Manifesto: four values and twelve principles
- Scrum: roles, artefacts, and events
- Extreme Programming (XP): key practices
- Kanban: visualising and managing flow
- Choosing a process model: a decision framework

Preparation
Read Sommerville (R2) Chapter 2 (agile sections) and Chapter 3. Also read the Agile Manifesto at agilemanifesto.org -- it is short and worth reading in full. Ensure your project group (3--4 students) is formed by the end of this week.

References

Bruegge, B. & Dutoit, A. (2013). Object-Oriented Software Engineering Using UML, Patterns, and Java. 3rd Edition. Prentice Hall.

Sommerville, I. (2016). Software Engineering. 10th Edition. Pearson.

Seidl, M., Scholz, M., Huemer, C. & Kappel, G. (2015). UML @ Classroom. Springer.

Pressman, R. & Maxim, B. (2020). Software Engineering: A Practitioner's Approach. 9th Edition. McGraw-Hill.

Royce, W.W. (1970). Managing the Development of Large Software Systems. Proceedings of IEEE WESCON.

IEEE Computer Society (2024). SWEBOK: Guide to the Software Engineering Body of Knowledge. Version 4.

ISO/IEC/IEEE 12207:2017. Systems and Software Engineering -- Software Life Cycle Processes.

Beck, K. et al. (2001). Manifesto for Agile Software Development. agilemanifesto.org.

ACM/IEEE-CS (1999). Software Engineering Code of Ethics and Professional Practice.