Software Development Processes

A process organises the activities that turn an idea into working software. This chapter presents the fundamental activities and the main process models, and teaches you to choose one for a project.

ISO/IEC/IEEE 12207:2017 | Waterfall · V · Incremental · Spiral · RUP | CMMI V3.0 | C++20 · CMake FetchContent · GoogleTest

AGENDA

What we will cover

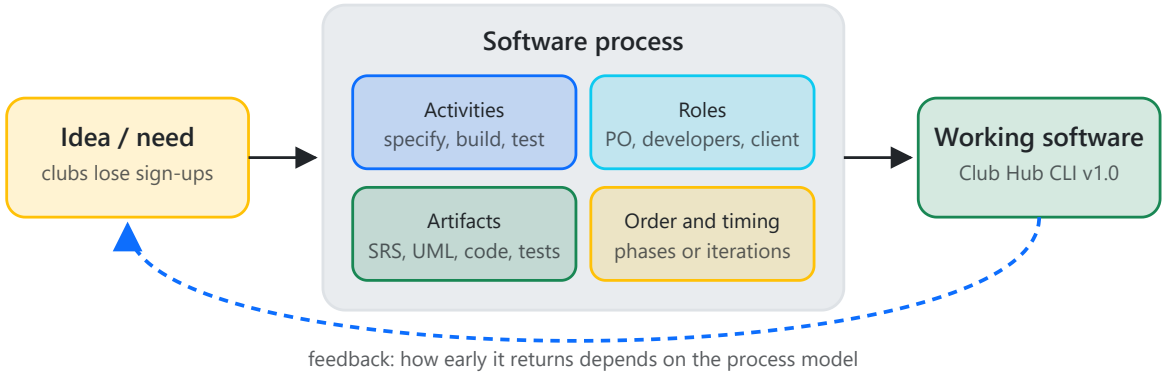
1. Fundamental activities: specification, design and implementation, validation, evolution
2. The software life cycle and ISO/IEC/IEEE 12207 vocabulary
3. Plan-driven versus agile development
4. The waterfall model and the V-model
5. Incremental and iterative development
6. Prototyping and the spiral model
7. The Rational Unified Process and its phases
8. Reuse-based and component-based development
9. Process improvement and maturity models (CMMI) at overview level
10. Choosing and tailoring a process for a project
11. Chapter Quiz
12. Lab-04

Click any item to jump directly to that topic.

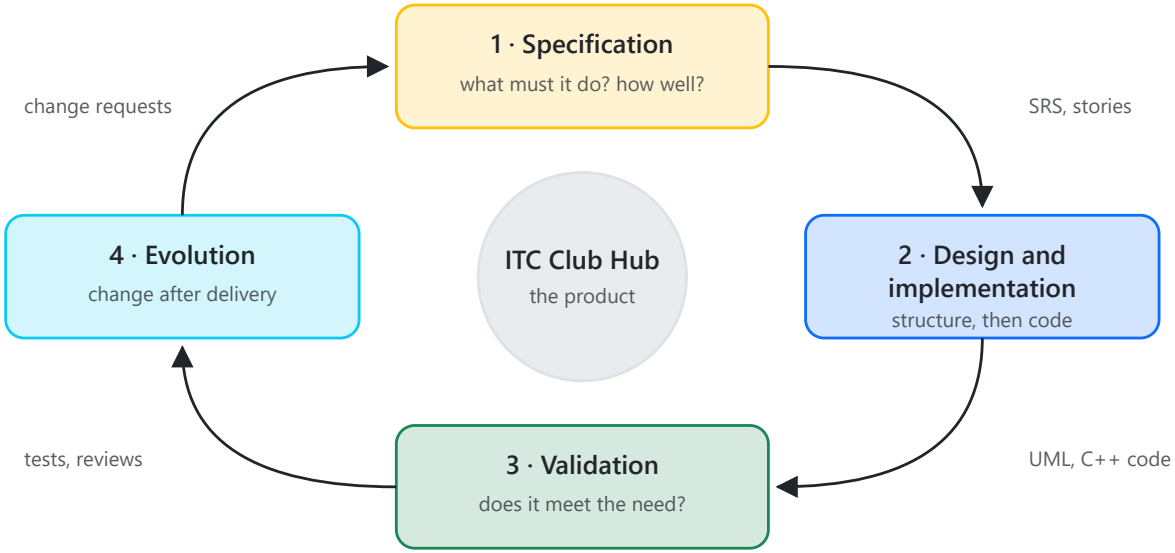
From idea to working software: why a team needs a process

- A **software process** is the structured set of activities a team follows to develop software: *who* does *what*, *when*, producing *which artifact*.
- A **process model** (waterfall, V-model, spiral, Scrum...) is a simplified, reusable description of a process. A team adapts a model into its own process.
- Five students building ITC Club Hub in 15 weeks already need answers: when do we write requirements? who tests? when does the client see something?
- There is **no best process** for every project. The skill you learn today is to *compare* models and *justify* a choice.

> 💡 Chapter 01 showed why software is hard (complexity, changing requirements, the cost-of-change curve). A process is the engineering answer: it decides when mistakes are found and how change is absorbed.



Four activities appear in every software process



- **Specification**: define what the system must do (functions) and how well (quality, constraints). Chapter 05.
- **Design and implementation**: decide the structure (architecture, classes) and turn it into code. Chapter 06 models, your C++ sources.
- **Validation**: show that the system does what the client needs: reviews, tests, demos. Chapter 09.
- **Evolution**: change the software as needs change after delivery. Chapter 10.

📖 Source: Sommerville, *Software Engineering* (10th ed.); SWEBOK Guide v4.0 covers each activity as a knowledge area. Process models differ only in **how they order and repeat** these four activities.

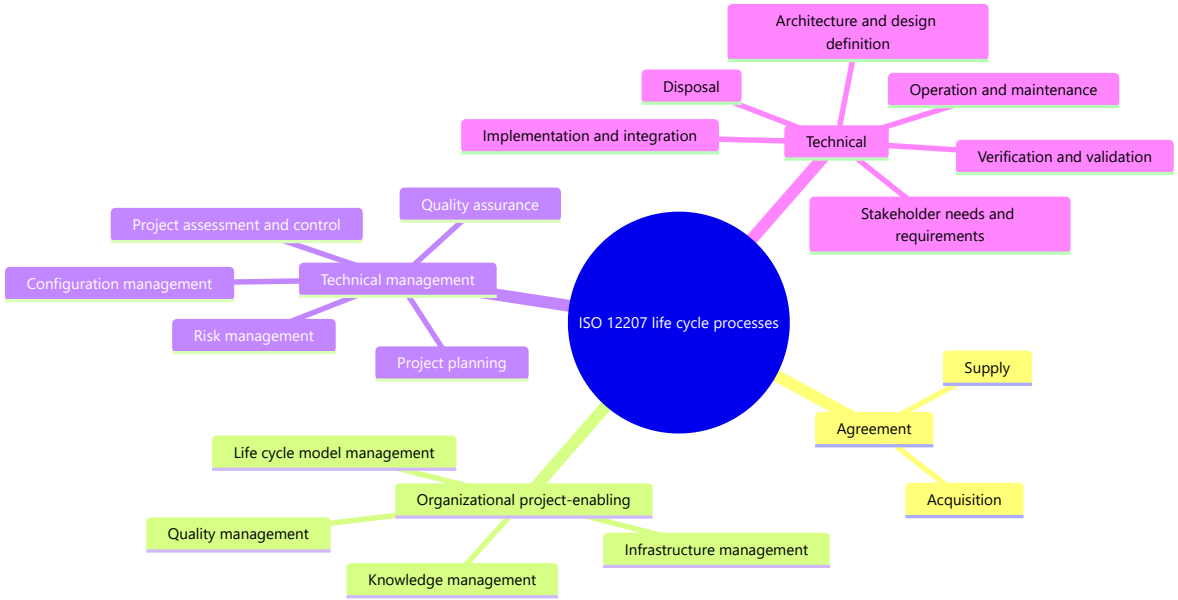
The four activities in the ITC Club Hub project

Activity	What happens in Club Hub	Who leads	Weeks	Artifact produced
Specification	Interview the client (the TA), write user stories such as "*As a student I want to register for an event so that I get a seat*", list rules: capacity, no double registration, cancel up to 24 h before start	Product Owner, whole team in workshops	5–6	`lab05/srs.md`, product backlog
Design and implementation	Model `Club`, `Event`, `Registration`; implement `RegistrationService::registerStudent` and CSV repositories in C++20	Developers	7, 9–13	UML diagrams, `src/`, `include/clubhub/`
Validation	GoogleTest unit tests for every business rule, peer review of pull requests, client demo at each sprint review	Developers, client at reviews	9–14	`tests/`, CI results, review notes
Evolution	Client asks for a waiting list after Sprint 1; the team handles it as a change request and releases v1.1	Product Owner + developers	11–14	Change request, `CHANGELOG.md`

⚠ **Misconception:** "the four activities are four phases done once". In most projects they overlap and repeat: validation of Sprint 1 already triggers evolution in week 11.

☑ **Good habit:** for each activity, name the *person*, the *week* and the *file*. If you cannot, the activity will probably not happen. You do exactly this in Lab-04 Task 1.

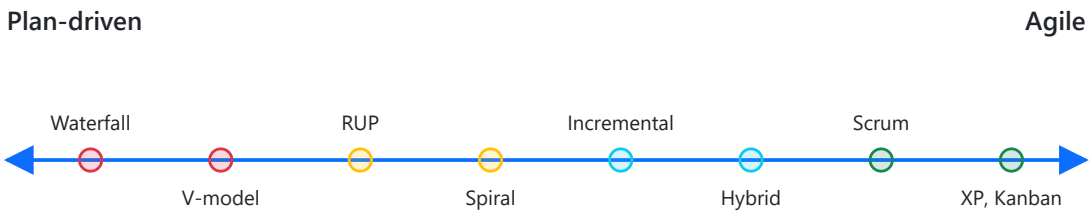
The software life cycle and the ISO/IEC/IEEE 12207 process groups



Term	Meaning (Club Hub example)
Life cycle	Evolution of a system from idea to retirement (Club Hub from week 1 until the club office stops using it)
Life cycle model	Framework of stages and their order (waterfall, spiral, Scrum-style iterations)
Process	Set of related activities with a purpose and outcomes (risk management)
Activity	Group of tasks inside a process (identify risks)
Task	Smallest required action (log risk R-03 "TA unavailable in week 8")
Work product	Artifact a task produces (`risks.md`)

📖 ISO/IEC/IEEE 12207:2017 defines 30 processes in four groups (simplified here). It says *what* processes exist, **not** in which order to run them: the order comes from the life cycle model you choose.

Plan-driven and agile are the two ends of a spectrum



Plan, then execute

- all activities planned in advance
- progress measured against the plan
- documents hand work between phases

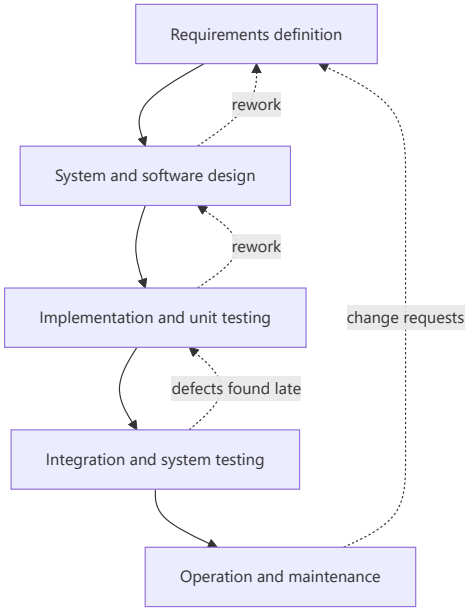
Plan a little, deliver, adapt

- planning is incremental, per iteration
- progress = working software
- conversation over heavy documents

Factor	Favours plan-driven	Favours agile
Requirements	stable, well known	unclear, changing
Criticality	safety, regulation	comfort, time to market
Client	available at milestones	available every week
Team	large, distributed	small, co-located
Contract	fixed scope and price	fixed time, flexible scope

📖 Boehm and Turner, *Balancing Agility and Discipline* (2003); Manifesto for Agile Software Development (2001). Most real projects are **hybrids**. Agile is covered in depth in Chapter 07.

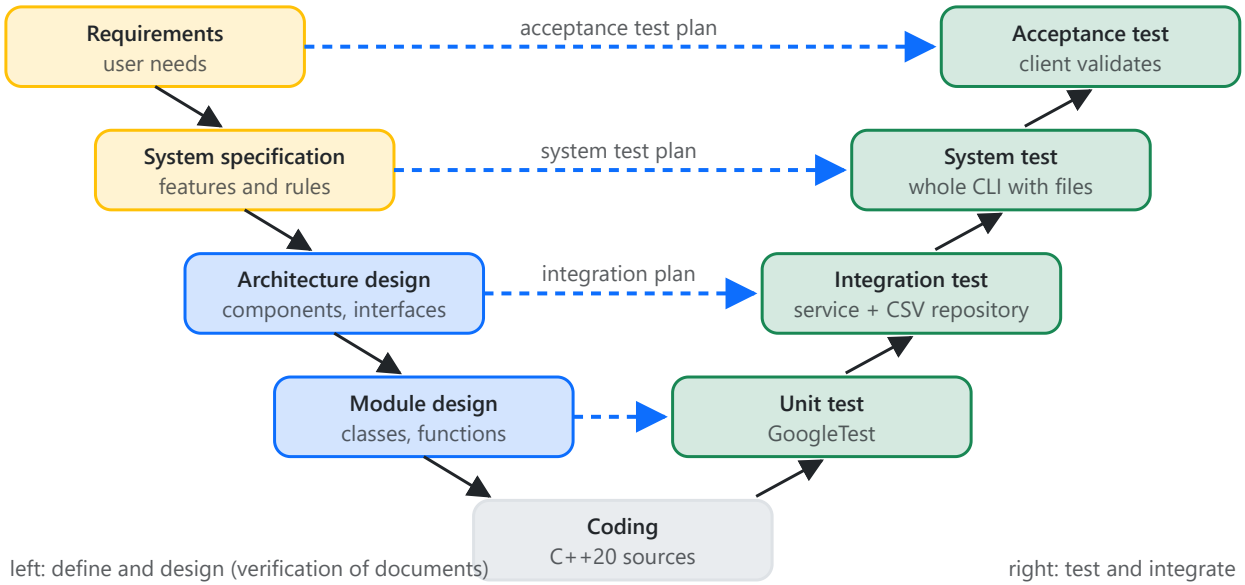
The waterfall model: one phase after the other



- Each phase produces **signed-off documents** that the next phase uses. In theory a phase starts only when the previous one is finished.
- The dashed **feedback arrows** are real: errors found late send work back up, which is expensive (the cost-of-change curve from Chapter 01).
- **Strengths**: easy to plan and to manage by milestones, strong documentation, fits contracts with fixed scope.
- **Weaknesses**: the client sees working software only at the end; changing requirements are painful; integration problems appear late.
- **Fits** when requirements are well understood and stable: a regulated payroll change, an embedded controller built with its hardware.

⚠ Royce's 1970 paper, often cited as the origin, already warned that the pure sequence is risky and recommended iteration and an early prototype.

The V-model: every design level has a matching test level



- The V-model is the waterfall **folded in the middle**: the left side goes down in detail, the right side goes up in integration.
- Each left-side artifact **defines the tests** of its mirror level before any code exists (dashed arrows).
- A failing test tells you *which level* is wrong: a failing integration test points back to the architecture design.
- Common in safety-critical and regulated domains (automotive, medical devices, railways) where traceability from requirement to test is mandatory.

> **Verification**: did we build the product right (against the spec)? **Validation**: did we build the right product (for the user)? Chapter 09 goes further.

V-model mapping: one Club Hub rule from requirement to test

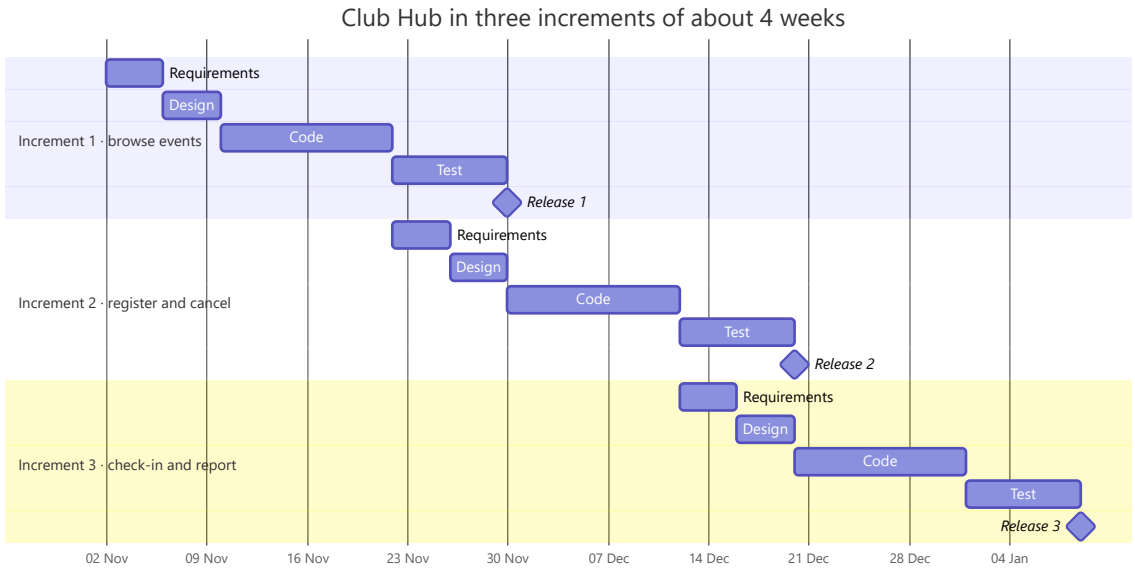
Left side (defines)	Club Hub artifact	Right side (tests)	Test for it
Requirement	REQ-07: "A student cannot register for an event that is full."	Acceptance test	AT-07: the client registers 30 students for a 30-seat event in the CLI; the 31st sees "Event full"
System specification	Rule: `activeRegistrations < capacity`, else error code `E_FULL`	System test	ST-07: script feeds CLI commands, checks output and `registrations.csv`
Architecture design	`RegistrationService` uses `EventRepository` and `RegistrationRepository`	Integration test	IT-07: service + `CsvRegistrationRepository` on a temp file, reload and count
Module design	`registerStudent(eventId, studentId)` throws `CapacityExceeded`	Unit test	UT-07: GoogleTest with in-memory fakes (right)

🖧 The shared number **07** is *traceability*: from any test you can find the requirement it proves, and from any requirement all its tests. Chapter 05 builds the full traceability matrix.

⚠ Writing all tests only after coding is the classic mistake. In the V-model the test *plans* are written on the way down; only their execution happens on the way up.

```cpp
// tests/registration_service_test.cpp
// Unit level of the V: one class, no files, fast.
#include <gtest/gtest.h>
#include "clubhub/RegistrationService.hpp"
// in-memory repositories and testEvent()
#include "fakes.hpp"

using namespace clubhub;

// UT-07 verifies REQ-07 at unit level
TEST(RegistrationServiceTest, RejectsWhenEventIsFull) {
    InMemoryEventRepository events;
    InMemoryRegistrationRepository regs;
    // an event with exactly one seat
    events.save(testEvent("E001", 1));
    RegistrationService service{events, regs};

    service.registerStudent("E001", "S001");

    EXPECT_THROW(service.registerStudent("E001", "S002"),
                 CapacityExceeded);
    EXPECT_EQ(regs.countActive("E001"), 1);
}
```

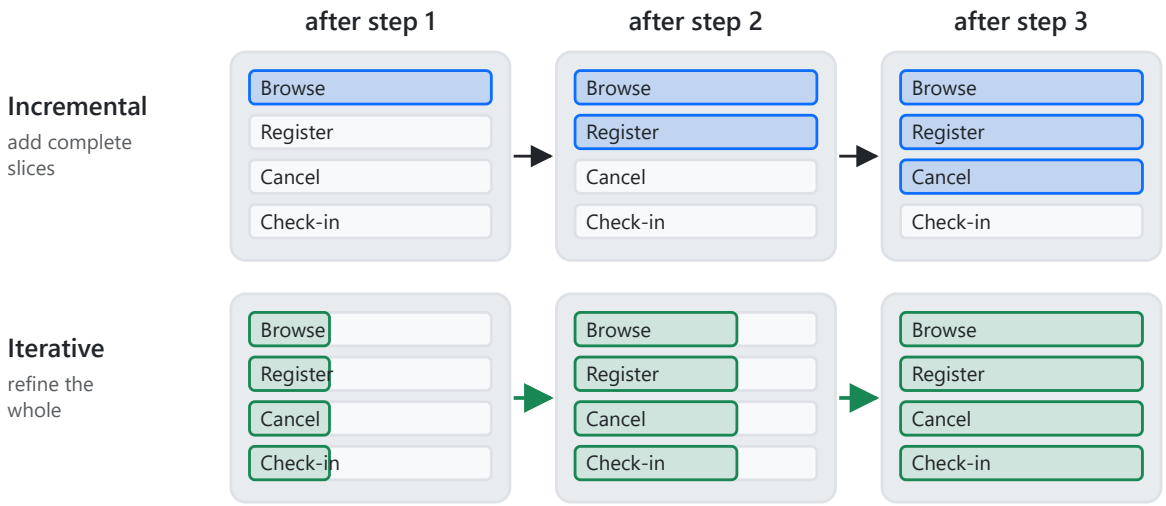
Incremental delivery: the system grows in complete slices



- The requirements are split into **increments**, most valuable first. Each increment is a mini-waterfall: requirements, design, code, test.
- After each increment the client gets **working, usable software** with part of the features.
- Planning the next increment overlaps the testing of the current one.
- **Benefits**: early value and feedback, lower risk of total failure, the most important features are tested the longest.
- **Risks**: the architecture must allow growth; without refactoring, structure degrades with each increment.

🖩 3 increments of 4 weeks each: the first working software arrives after **4** weeks instead of after 12 with waterfall. Overlapping the next increment's requirements with the current tests saves a few more days.

Iterative development: build it roughly, then refine it

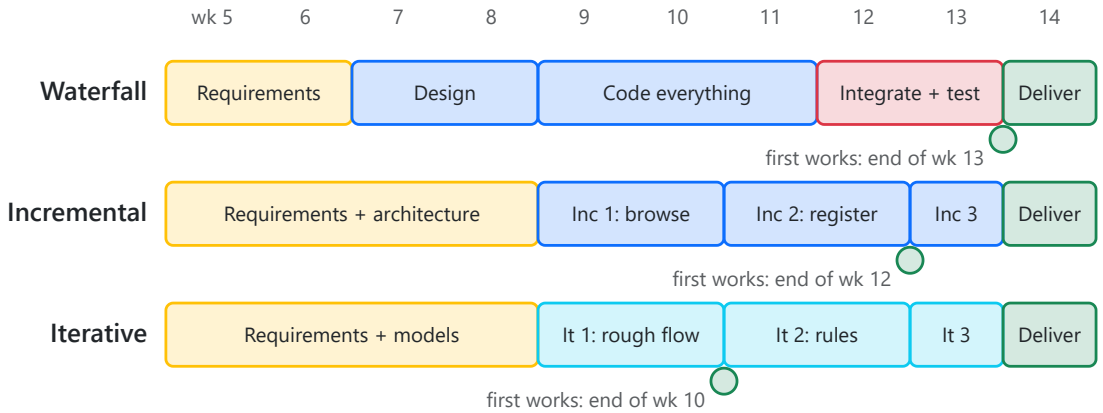


green arrows: client feedback on the whole product changes the next iteration

- **Incremental**: add finished pieces. Each piece is complete when delivered.
- **Iterative**: repeat the cycle on the *same* functionality, improving it each time using feedback. Early versions are deliberately rough.
- In practice they are combined: Scrum delivers an *increment* every sprint and *iterates* on features across sprints.
- Iteration 1 for "register for an event": CLI flow works end to end, no rules. Iteration 2: capacity and no double registration. Iteration 3: 24-hour cancellation rule, waiting list, reminder.

Iterative development attacks **uncertainty** (we are not sure what the client wants); incremental delivery attacks **time to value** (the client needs something early).

"Register for an event" through three processes: when does it work?

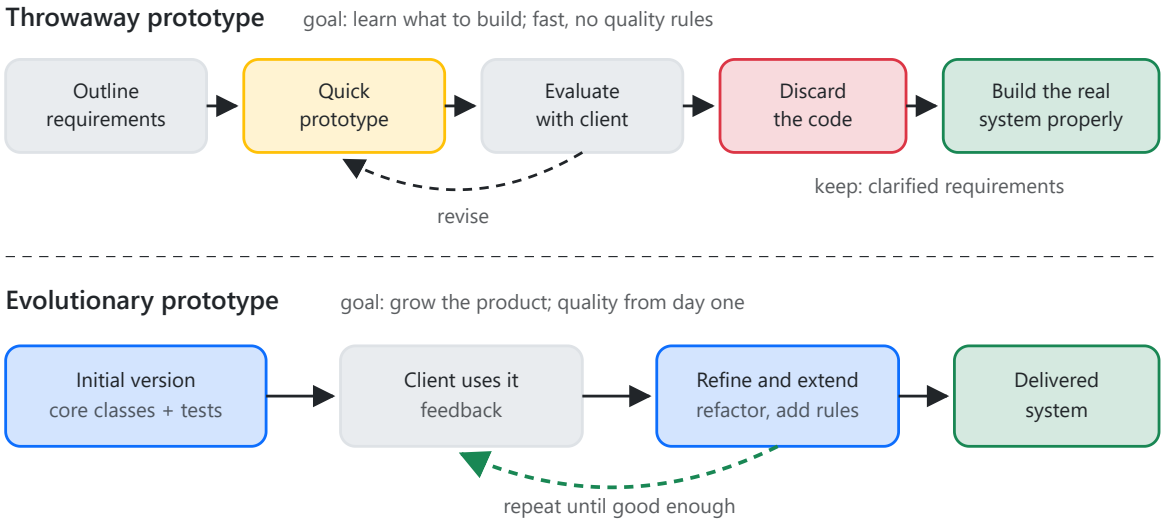


Process	Client first registers	Weeks left to react
Waterfall	end of week 13	1 (only fixes)
Incremental	end of week 12	2 (browse seen in week 10)
Iterative	end of week 10, rough	4 (two more iterations)

- Suppose the client says at the first demo: "students must see how many seats are left before they register". With waterfall this surprise arrives in week 13; with iteration 1 it arrives in week 10 and costs one extra column in the event list.
- Weeks 5–8 are the same in all three: requirements (Chapter 05) and models (Chapter 06) come before the course's sprints.

☑ The earlier the first working version, the cheaper each surprise. This is the main argument for incremental and iterative processes.

Prototyping: throwaway versus evolutionary



- A **prototype** is an early, partial version used to try out ideas, answer questions and reduce uncertainty.
- **Throwaway**: built only to learn (screen flow, wording, a risky algorithm). Hard-coded data, no tests. After the client session the code is discarded; the *lessons* go into the requirements.
- **Evolutionary**: built with production quality from the start and grown into the final product. This is essentially iterative development.
- Other forms: paper sketches, clickable UI mock-ups (Figma), a *spike* that tests one technical question.

⊗ **Classic trap:** the client likes the throwaway demo and asks "can we ship it next week?". Say no: it has no tests, no error handling and no structure. Decide the prototype type *before* you build it.

A 15-line throwaway prototype for the registration screen

```cpp
// prototype/mock_register.cpp
// THROWAWAY: shown to the client once, then deleted.
#include <iostream>
#include <string>

int main() {
    std::cout << "ITC Club Hub (mock)\n"
              << "1) Robotics Workshop  Sat 14:00  12/30 seats\n"
              << "2) Hackathon Night    Fri 18:00  30/30 seats\n"
              << "Register for event #: ";
    int choice = 0;
    std::cin >> choice;
    const std::string reply = (choice == 1)
        ? "Registered! Reminder 24 h before the start.\n"
        : "Sorry, this event is full. Join waiting list? (y/n)\n";
    std::cout << reply;
}
```

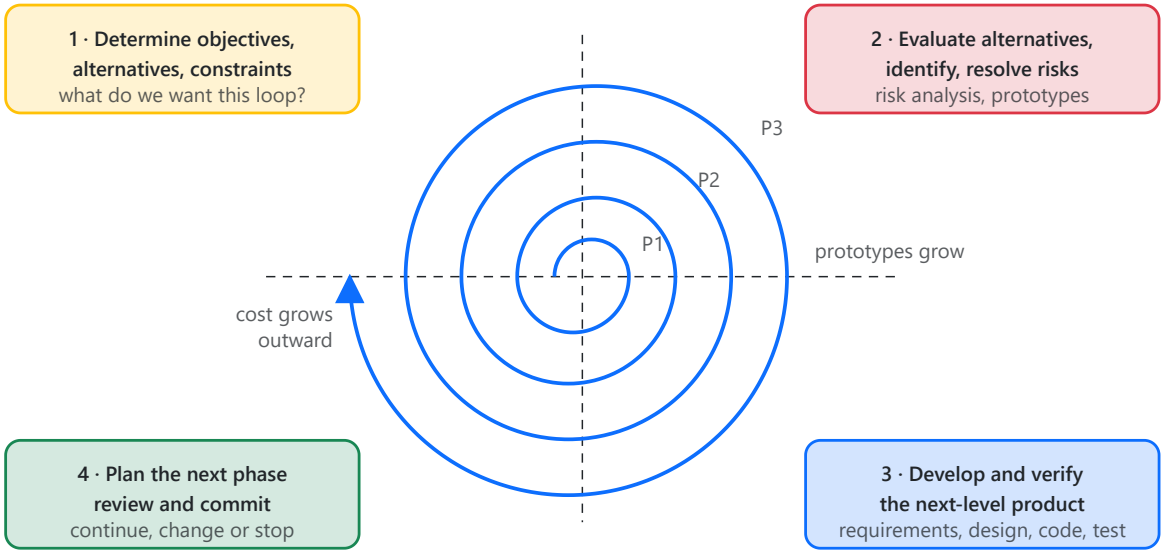
ⓘ Questions this mock answers in a 10-minute client session: Is a numbered list enough? Should seats left be shown? Does the client want a waiting list (a Should feature)? Is a reminder message expected?

	Throwaway mock (left)	Evolutionary version
Data	hard-coded strings	`Event` objects loaded by `CsvEventRepository`
Rules	faked by `if (choice == 1)`	`RegistrationService` checks capacity, duplicates, deadline
Tests	none	GoogleTest for every rule
Time to build	20 minutes	a sprint
After the demo	deleted, lessons written into user stories	kept, refined in the next iteration
Good for	unclear UI and wording	clear core, uncertain details

- Keep throwaway code in a clearly named folder (`prototype/`) that is **not** part of the CMake build, and delete it after the session.
- Record what you learned: "client wants seats left in the list" becomes a new acceptance criterion in Chapter 05.

⚠ Notice what the mock does *not* have: no `Event` class, no validation of input, no error handling. That is fine for learning and unacceptable for the product.

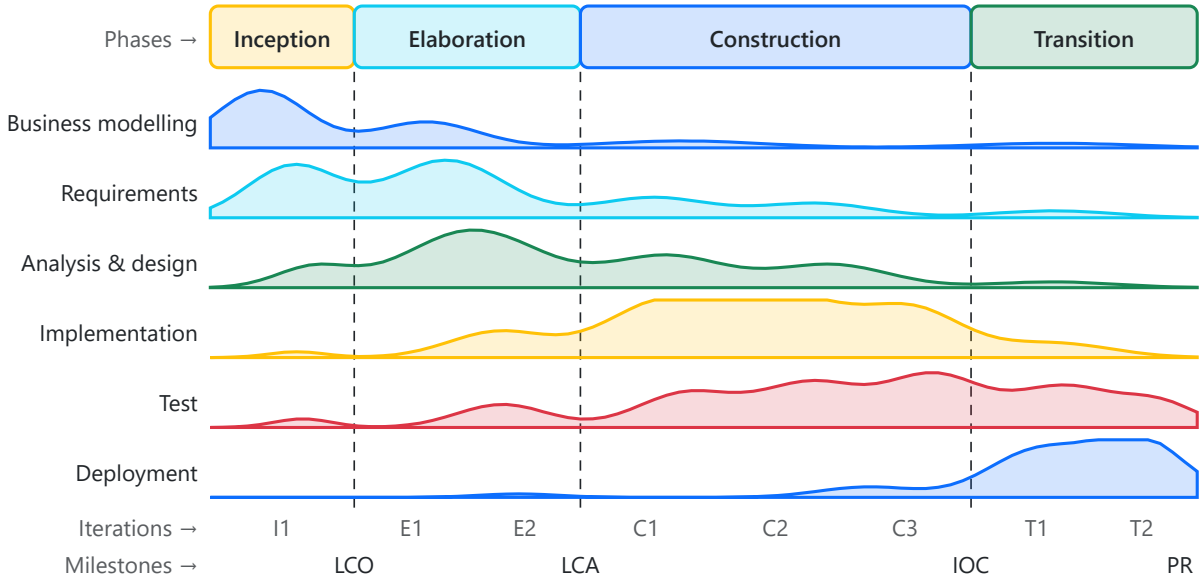
The spiral model: every loop starts from the biggest risk



- Proposed by Barry Boehm (1988). Instead of fixed phases, the project runs **loops**; each loop passes through four quadrants.
- The key quadrant is **risk analysis**: the team asks "what could make this project fail?" and resolves the top risk first, often with a prototype.
- The radius shows cumulative cost; the review at the end of each loop is a **go / no-go** decision.
- The spiral is a *meta-model*: a loop may look like a waterfall, a prototype or an increment, whatever reduces the risk best.

Club Hub risk	Loop that resolves it
Client unsure what "register" shows	Loop 1: throwaway CLI mock
CSV file corrupted if the program crashes mid-write	Loop 2: spike, write temp file then rename
Team new to CMake and GoogleTest	Loop 2: walking skeleton with one test in CI

RUP: four phases in time, disciplines that overlap



- The **Rational Unified Process** (Kruchten, 2003) is iterative and use-case driven. It has two dimensions: **phases** in time and **disciplines** of work.
- Each row of the "hump chart" shows how much effort one discipline takes over time. Requirements work peaks early but never drops to zero; testing grows during construction.
- Every phase contains one or more **iterations**, each producing an executable release.
- Supporting disciplines (configuration and change management, project management, environment) run throughout and are omitted here.

💡 Humps are illustrative, not measured data. The idea to remember: **all disciplines happen in every phase**, only their weight changes.

RUP phases, their milestones and the Club Hub semester

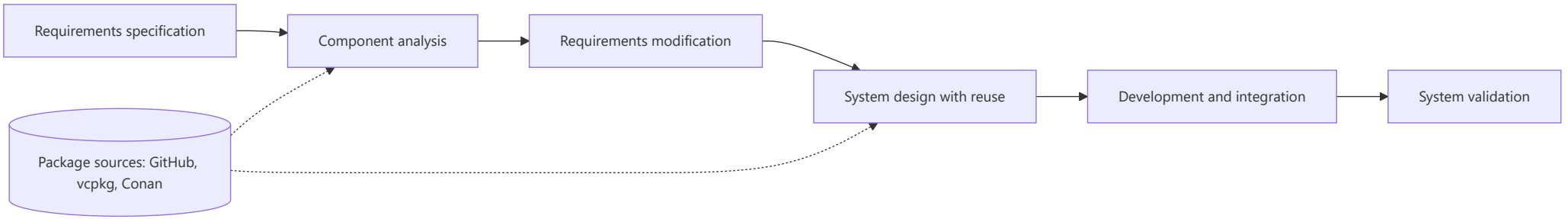
Phase	Main goal	Milestone at the end	Club Hub equivalent
Inception	Business case, scope, key risks, first estimate	**LCO** · Lifecycle Objectives: do we go on?	Weeks 1–4: team charter, problem statement, application type, ethics review, process decision (Lab-01 to Lab-04)
Elaboration	Stable, executable *architecture*; most requirements understood; main risks removed	**LCA** · Lifecycle Architecture	Weeks 5–8: SRS and backlog, UML models, a walking skeleton (CMake, one test, CI); checkpoint in week 8
Construction	Build the remaining features iteratively, test them	**IOC** · Initial Operational Capability: beta ready	Weeks 9–12: Sprint 1 and Sprint 2
Transition	Deploy to users, fix, train, hand over	**PR** · Product Release	Weeks 13–14: hardening, submission and demo

✨ Six RUP best practices

1. Develop iteratively
2. Manage requirements
3. Use component-based architectures
4. Model software visually (UML)
5. Verify quality continuously
6. Control changes to software

⚠ RUP was a heavy commercial product with many roles and artifacts; few teams used all of it. Its **phase and milestone idea** survives in hybrid processes, including the one your course follows.

Reuse-based development: search before you write



- **Component analysis**: search for libraries that cover a requirement. **Requirements modification**: sometimes adapt a requirement so an existing component fits (accept its CSV dialect instead of inventing ours).
- Reuse happens at many levels: *functions and libraries* (C++ standard library, GoogleTest), *frameworks* (Qt), *components and services* (an email API), *whole systems* (COTS, off-the-shelf products).
- **Component-based development**: build the system from parts that talk only through *interfaces*, so one implementation can replace another: `CsvEventRepository` today, maybe `JsonEventRepository` later, both behind `EventRepository` (right).
- Benefits: faster, tested code, standards. Costs: learning, dependency updates, licence obligations (Chapter 03), less control.

```cpp
// include/clubhub/EventRepository.hpp
// A component interface: callers never see the storage.
#pragma once
#include <chrono>
#include <optional>
#include <string>
#include <vector>
#include "clubhub/Event.hpp"

namespace clubhub {
class EventRepository {
public:
    virtual ~EventRepository() = default;
    virtual std::optional<Event> findById(
        const std::string& id) const = 0;
    virtual std::vector<Event> findUpcoming(
        std::chrono::sys_seconds now) const = 0;
    virtual void save(const Event& event) = 0;
};
}  // namespace clubhub
```

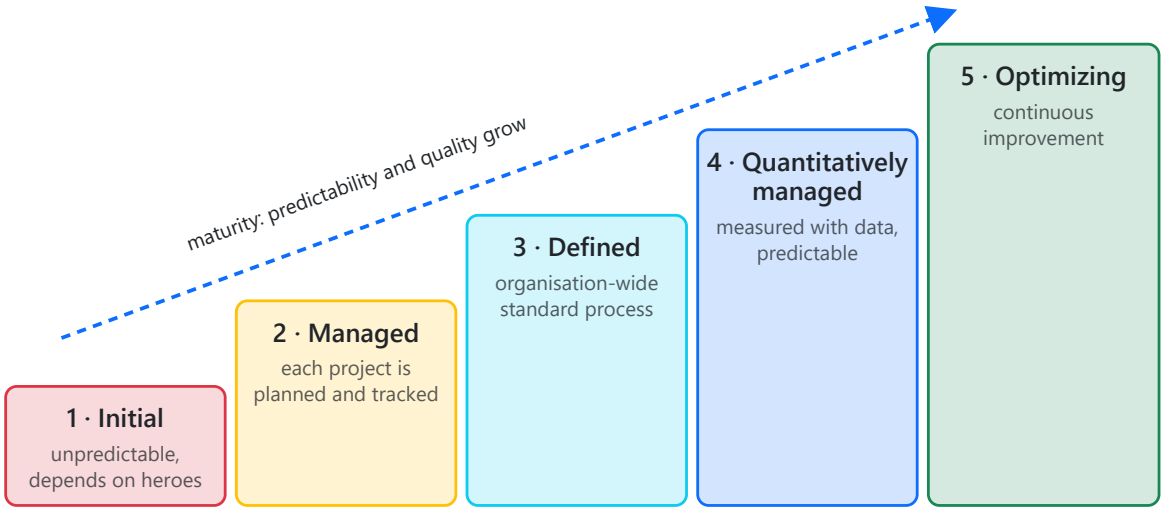
Make or reuse? GoogleTest and a CSV library via CMake FetchContent

```
# CMakeLists.txt (excerpt): reused components, pinned versions
include(FetchContent)

# Reuse 1: GoogleTest (BSD-3-Clause), unit-test framework
FetchContent_Declare(googletest
  URL https://github.com/google/googletest/archive/refs/tags/v1.17.0.zip
  DOWNLOAD_EXTRACT_TIMESTAMP TRUE)
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)

# Reuse 2: rapidcsv (BSD-3-Clause), header-only CSV reader
FetchContent_Declare(rapidcsv
  GIT_REPOSITORY https://github.com/d99kris/rapidcsv.git
  GIT_TAG        v8.84
  GIT_SHALLOW    TRUE)

FetchContent_MakeAvailable(googletest rapidcsv)

# Built by us: the business rules are Club Hub's own value
add_library(clubhub_core
  src/RegistrationService.cpp src/CsvEventRepository.cpp)
target_include_directories(clubhub_core PUBLIC include)
target_link_libraries(clubhub_core PUBLIC rapidcsv)

add_executable(clubhub_tests tests/registration_service_test.cpp)
target_link_libraries(clubhub_tests
  PRIVATE clubhub_core GTest::gtest_main)
```

Need	Decision	Reason
Unit tests	**Reuse** GoogleTest	mature, CTest and CI support, course standard
CSV parsing	**Reuse** rapidcsv	quoted fields and header lookup already solved; BSD-3 fits an MIT repo
Dates, 24 h rule	**Reuse** `std::chrono`	standard library, no dependency
Registration rules	**Build**	unique to Club Hub; this is what we are graded on
CLI menu	**Build**	about 50 lines with `std::getline`
E-mail reminder	**Defer**	print to console now; SMTP library later

📌 Always **pin a version** (tag or archive URL). `GIT_TAG main` means tomorrow's build may differ from today's. Check the newest release tag when you add a dependency.

CMMI: five maturity levels of an organisation's process



- **CMMI** (Capability Maturity Model Integration, current version V3.0, 2023, CMMI Institute) grew from the SEI Capability Maturity Model of the late 1980s.
- It describes **practice areas** (planning, requirements development, verification, configuration management, peer reviews, ...) and groups them into **maturity levels**.
- An organisation is **appraised** by certified appraisers; many government and outsourcing contracts ask for level 3 or higher.
- Each level builds on the one below: you cannot measure (4) a process you have not defined (3).

📖 CMMI tells you *what* good practice looks like, not *how* to do it: it works with waterfall and with Scrum. Related: ISO/IEC 33001 (process assessment). Overview level only in this course.

Process improvement in a student team: measure, analyse, change

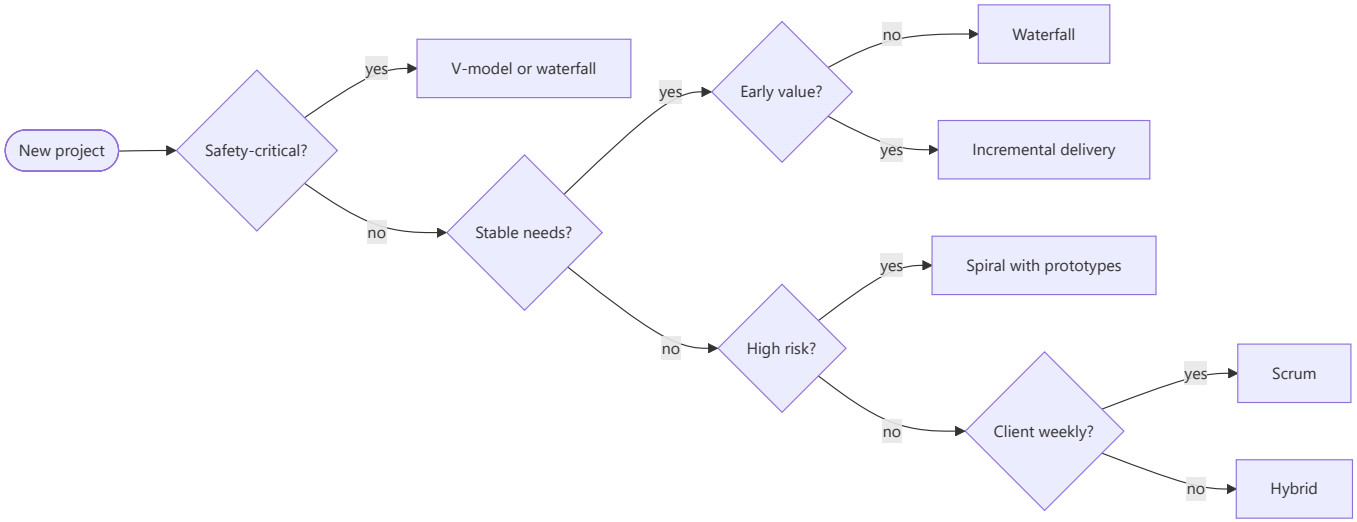
Level	What it looks like in a Club Hub team	Evidence you could show
1 · Initial	One strong student codes all night before the deadline; nobody else can build the project	one author in `git log`
2 · Managed	Backlog and board kept up to date, work estimated and tracked, every file in Git with branches and tags	board history, tags `lab-04`, `v0.1`
3 · Defined	Written Definition of Done, pull-request template and review checklist used by every member	`CONTRIBUTING.md`, reviewed PRs
4 · Quantitatively managed	Velocity, CI pass rate and defects per sprint recorded and used to plan Sprint 2	`metrics.csv`, burndown
5 · Optimizing	Each retrospective picks one change, sets a target and checks it next sprint	retro actions with results

The improvement loop

1. **Measure**: "4 of 9 pull requests in Sprint 1 were merged without review."
2. **Analyse**: reviews were forgotten, not refused; no rule enforced them.
3. **Change**: protect `main`, require one approval (Chapter 08).
4. **Check**: Sprint 2 shows 0 unreviewed merges. Keep the change.

A team cannot "be CMMI level 3" in one semester, but it can adopt single practices now: configuration management, peer reviews, a written Definition of Done. The Lab-04 challenge asks you to pick one.

Choosing a process: a decision guide



- Read the questions in full: *safety-critical* or fixed-scope contract? requirements *stable* and understood? *early value* needed before the end? *high* technical or market risk? client available *every week* or two? *Hybrid* = fixed milestones with internal iterations.
- A decision guide is a **starting point**, not a rule. Real projects mix answers: a V-model for the safety part, Scrum for the user interface.
- Score options against **explicit criteria**: requirements stability, client availability, team experience, deadline, risk, criticality, team size.
- Write the choice down as a **process decision record**: context, options considered, decision, consequences. Anyone joining later understands *why*.

Club Hub fact	Points to
Requirements still vague in week 4	iterative
TA available 30 min each week	iterative
Fixed midterm, submission in week 14	fixed milestones
Graded SRS and UML	some up-front documents

Tailoring: a process record for a 5-person team over 15 weeks

```
# Process tailoring record · Team Mekong Coders · v1.0 (week 4)
Base model: Scrum (Scrum Guide 2020) inside fixed course milestones
Context: 5 students · 15 weeks · about 6 h per person per week
         client = teaching assistant, available Fridays for 30 min

| Element       | Reference practice  | Our tailoring                 |
|---------------|---------------------|-------------------------------|
| Up-front work | emergent            | weeks 5-8: SRS, UML, skeleton |
| Sprint length | one month or less   | 2 weeks: wk 9-10 and 11-12    |
| Daily Scrum   | every day, 15 min   | Mon/Wed/Fri 10 min + chat log |
| Product Owner | one person          | one student; the TA is client |
| Scrum Master  | one person          | rotates every sprint          |
| Documents     | only what is useful | SRS and UML are graded: keep  |
| Hardening     | not in Scrum        | week 13: fixes only, freeze   |
| Done means    | team decides        | CI green, PR reviewed, demoed |

Why: exam weeks and graded documents need fixed milestones;
vague requirements and a weekly client need short iterations.
Review: at every retrospective; changes go in the change log.
```

- **Tailoring** means adapting a process model to the project: adding, removing or changing activities, artifacts, roles, cadence and formality. ISO/IEC/IEEE 12207 expects it.
- Tailor for a **reason** written next to each change. "We skipped testing because we were busy" is not tailoring.
- Keep the **essentials**: for Scrum these are the Sprint, the Product Owner, the reviews and a Definition of Done.

Capacity check for one sprint

5 students × 6 h × 2 weeks = **60 h**. Minus about 20 % for Scrum events and reviews = **48 h** for stories. At roughly 4 h per small story: plan about **12 stories**, not 25.

This record is what you produce in Lab-04 Tasks 3 and 4. Chapter 07 then runs the sprints for real.

WRAP-UP

Best practices and common mistakes

1 Choose with criteria

Score models against your project's facts and record the decision. Habit ("we always do Scrum") is not a reason.

2 Working software early

Aim for a rough end-to-end version as soon as possible; every surprise found early is cheaper.

3 Test levels mirror design

Even in agile, keep the V-model idea: every requirement has an acceptance test, every class has unit tests.

4 Reuse with care

Reuse mature libraries, pin their versions, check their licence, and build only what makes your product unique.

✕ Waterfall in disguise

"Sprints" in which all testing is left to the last week. Calling phases sprints does not make a process iterative.

✕ Prototype becomes product

Throwaway code with hard-coded data shipped because the demo looked good. Decide the prototype type first.

✕ "Agile means no documents"

Agile values working software *more*, not documents *never*. Keep the SRS, models and decisions you need.

✕ Unwritten tailoring

Silently dropping reviews or tests under pressure. Every tailoring decision needs a reason and a place in the record.

Chapter quiz 10 questions

1. Which of these is not one of the four fundamental software process activities?

- Specification
- Validation
- Evolution
- Marketing

2. In ISO/IEC/IEEE 12207:2017, configuration management belongs to which process group?

- Agreement processes
- Organizational project-enabling processes
- Technical management processes
- Technical processes

3. The GoogleTest case `RejectsWhenEventIsFull` checks one member function of `RegistrationService` with in-memory fakes. Which left-side V-model artifact does it verify?

- Requirements
- System specification
- Architecture design
- Module design

4. Club Hub is built in 3 increments of 4 weeks each (12 weeks in total). With waterfall, working software appears at the end of week 12. With incremental delivery, when does the client first get working software?

- End of week 1
- End of week 4
- End of week 8
- End of week 12

5. In weeks 9 and 10 a team shows a registration flow that works end to end but ignores capacity and cancellation rules; in weeks 11 and 12 they add the rules to the same flow after client feedback. Which approach is this?

- Waterfall
- Iterative development
- Throwaway prototyping
- Reuse-based development

6. After the client session, what should happen to the 15-line C++ CLI mock with hard-coded events?

- Refactor it into the product
- Delete it and write the lessons into the requirements
- Ship it as release 1
- Add it to the CMake build so CI compiles it

7. What distinguishes the spiral model from the other process models?

- A fixed sequence of phases with sign-off
- An explicit risk analysis in every loop decides what to do next
- Time-boxed two-week sprints
- Building the system from off-the-shelf components

8. Which RUP phase has as its main goal a stable, executable architecture with the main risks removed?

- Inception
- Elaboration
- Construction
- Transition

9. Each Club Hub team plans and tracks its own work with a backlog and a board, but every team works differently and there is no shared standard process. Which CMMI maturity level fits best?

- Level 1 · Initial
- Level 2 · Managed
- Level 3 · Defined
- Level 4 · Quantitatively managed

10. In a make-or-reuse analysis, which Club Hub component should the team build rather than reuse?

- A unit-testing framework
- A CSV parser
- The rule that a student cannot register twice for the same event
- Date and time arithmetic for the 24-hour rule

WRAP-UP

Summary

Every process organises four activities: specification, design and implementation, validation and evolution. Models differ in how they order and repeat them.

ISO/IEC/IEEE 12207 gives the vocabulary (life cycle, process, activity, task) and four process groups, but no fixed order.

Plan-driven and agile are ends of a spectrum; choose by requirements stability, client availability, criticality and team.

Waterfall suits stable requirements; the V-model pairs each design level with a test level and gives traceability.

Incremental delivery adds complete slices; iterative development refines the whole. Both bring working software and feedback early.

Throwaway prototypes answer questions and are deleted; evolutionary prototypes grow into the product; the spiral model tackles the biggest risk in each loop.

RUP has four phases (Inception, Elaboration, Construction, Transition) with milestones; its disciplines overlap in every iteration.

Reuse mature components such as GoogleTest and a CSV library through pinned FetchContent dependencies; build what is unique.

CMMI describes five maturity levels; a team improves by measuring, analysing and changing one practice at a time, and records its tailoring.

Open Lab-04: 5 tasks + 1 challenge

Next chapter: 05 · Requirement Engineering