Software Development Challenges

Software is everywhere, and building it well is hard. This chapter introduces software engineering as a discipline, explains the essential and accidental difficulties of building software, and uses real failures to show what is at stake.

SWEBOK Guide v4.0 | Brooks, No Silver Bullet (1986) | C++20 · CMake · GitHub | Case study: ITC Club Hub

AGENDA

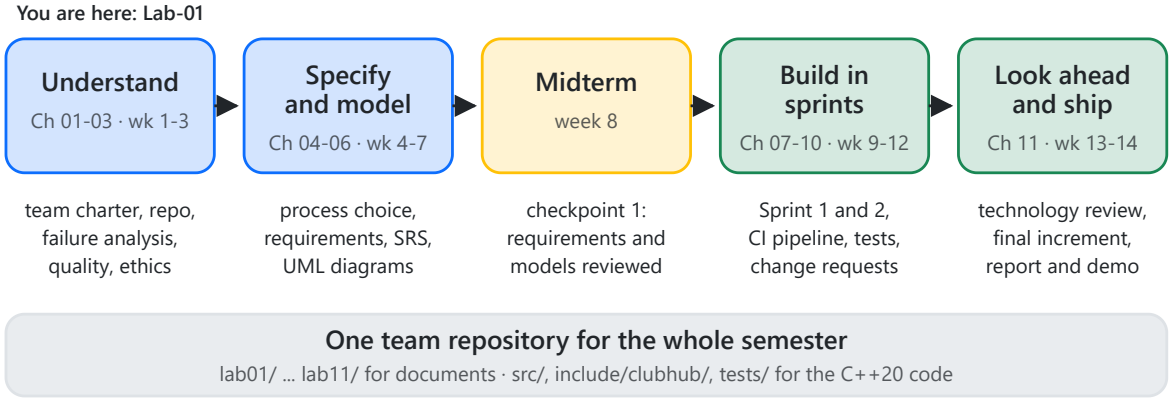
What we will cover

1. What software engineering is: programs versus software products, and the engineering mindset
2. A short history: the software crisis and the birth of the discipline
3. Essential difficulties: complexity, conformity, changeability, invisibility (Brooks)
4. Scale: from a single developer to large teams, and the cost of communication
5. Changing requirements and the cost-of-change curve
6. Reliability, security and safety expectations of modern software
7. Legacy systems, integration and technical debt
8. Case studies of software failures (Therac-25, Ariane 5, Knight Capital, a recent outage)
9. The SWEBOK knowledge areas as a map of the discipline and of this course
10. Roles in a software team: developer, tester, business analyst, product owner, UX designer, DevOps engineer
11. Chapter Quiz
12. Lab-01

Click any item to jump directly to that topic.

From "my program works" to "our product works for everyone"

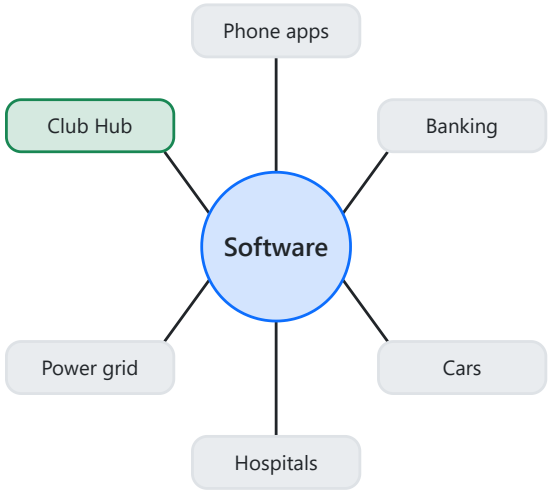
- You can already write a C++ program that works on your laptop. This course is about building **software that other people depend on**, with a team, over months and years.
- Each chapter adds one engineering practice; each lab applies it to one team project: **ITC Club Hub**, an application for student clubs at the institute.
- Teams of 4 to 5 analyse and model the full web and mobile system and implement its core in **C++20**: domain model, business rules, a command-line front end, CSV or JSON storage.
- This chapter asks the first question: *why is building software hard, and what goes wrong when we ignore that?*

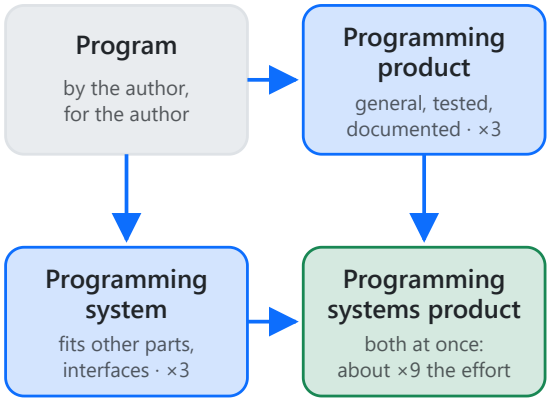


Software is everywhere, and a product is much more than a program

- **Software engineering**: the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software (IEEE definition, used by SWEBOK Guide v4.0).
- A **program** is written by its author for its author. A **software product** is used by people who did not write it, on machines the author never saw, for years.
- Brooks (The Mythical Man-Month, 1975): making a program general, tested and documented triples the effort; making it fit into a larger system triples it again, so about **9 times** in total.
- **Engineering mindset**: understand the problem before coding, make trade-offs explicit (time, cost, quality), measure instead of guessing, design for change, and take responsibility for the result.

Software everywhere





Brooks, The Mythical Man-Month (1975)

Worked example: the same feature as a program and as a product

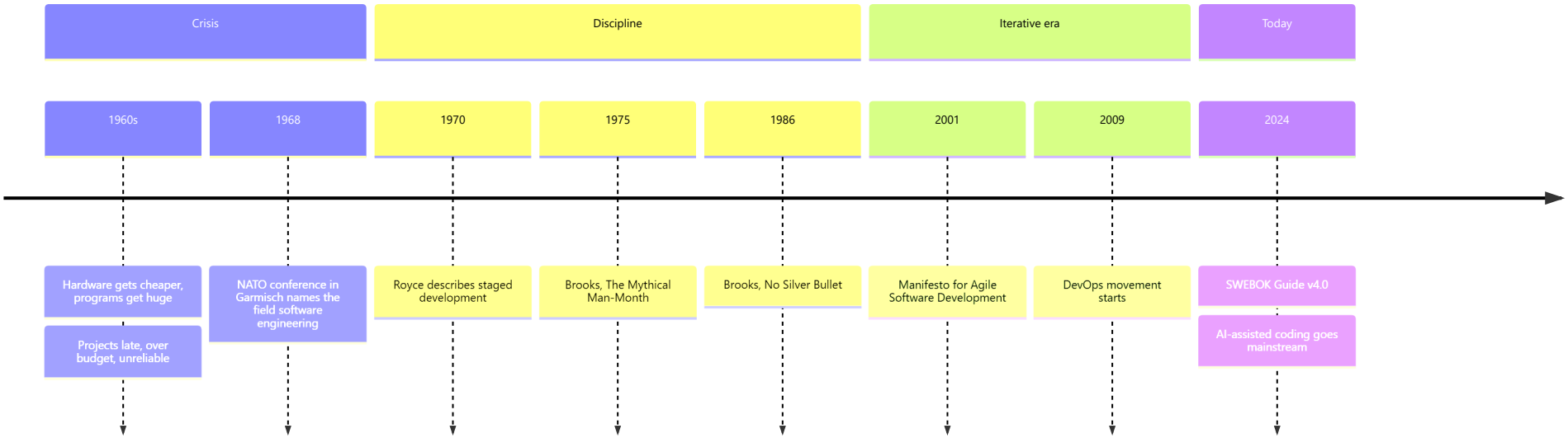
```cpp
// A program: works for its author, today, on this laptop
int main() {
    int capacity, registered;
    std::cin >> capacity >> registered;
    std::cout << capacity - registered << " seats left\n";
}
```

```cpp
// A product: other users, bad input, future change, tests
namespace clubhub {

// Seats still free for an event. Never negative: an overbooked
// event reports 0, and the overbooking itself is caught elsewhere.
[[nodiscard]] constexpr int seatsLeft(int capacity, int registered) {
    if (capacity < 0 || registered < 0) {
        throw std::invalid_argument("negative count");
    }
    return std::max(0, capacity - registered);
}

}  // namespace clubhub

// tests/seats_left_test.cpp (GoogleTest, Chapter 09)
TEST(SeatsLeft, NeverNegative) {
    EXPECT_EQ(clubhub::seatsLeft(30, 32), 0);
}
```

Question	Program	Product
Who uses it?	The author	Hundreds of students, admins
Bad input?	"I will type it right"	Validated, clear error
How long does it live?	Until the assignment is graded	Years, many releases
Who changes it?	The author	People who never met the author
How do we know it works?	"It ran once"	Automated tests, reviews
Where are the rules written?	In one head	Requirements, docs, tests

The extra lines on the right are not "bureaucracy": each one answers a question that a user, a tester or a future maintainer will ask. That is the ×3 of Brooks's grid in miniature.

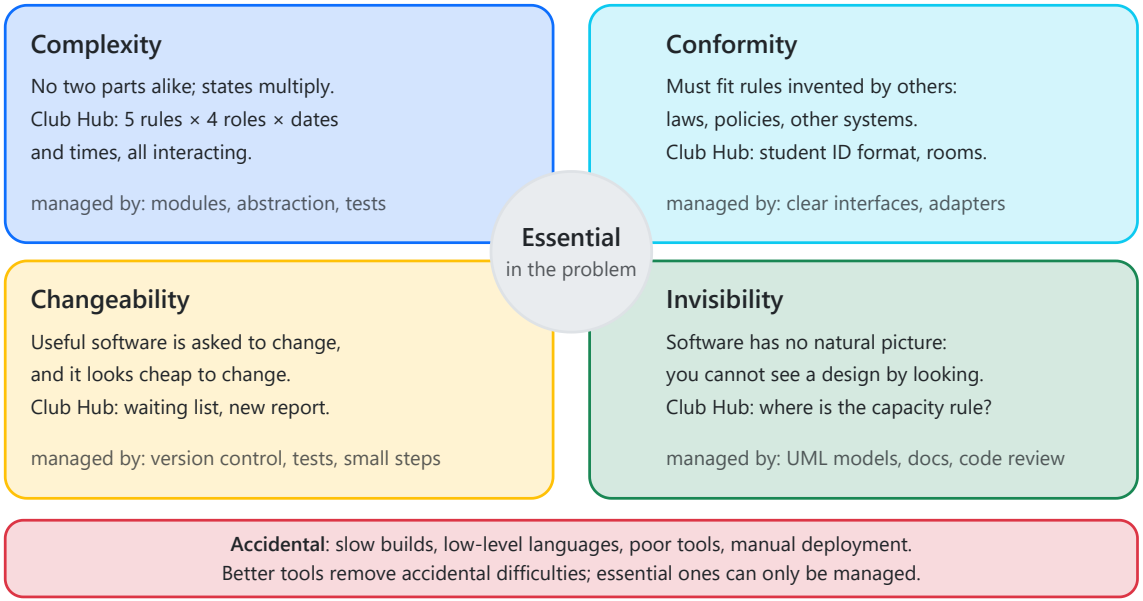
The software crisis and the birth of the discipline



- In the 1960s computers became powerful enough for large systems (operating systems, airline booking, defence), and projects failed in ways nobody had planned for: this became known as the **software crisis**.
- The **1968 NATO conference** chose the provocative title "Software Engineering": building software should rest on the theory and discipline that other engineering fields already had.
- Each generation since then found new answers: structured programming, process models, object orientation, agile, DevOps, cloud. The crisis never fully ended, because systems grew as fast as our methods improved; the failures in Topic 8 all happened after 1985.

📖 SWEBOK Guide v4.0 (IEEE Computer Society, 2024) maps what the discipline knows today; Topic 9 uses it as the map of this course.

Why there is no silver bullet: four essential difficulties



- Brooks, **No Silver Bullet (1986)**: the hard part of software is the *essence*, the concepts and their relations, not the *accidents* of writing them down in a language on a machine.
- High-level languages, time-sharing and IDEs removed large accidental difficulties. No single new tool can make the essence ten times easier: hence "no silver bullet".
- Each essential difficulty has practices that **manage** it; they are the chapters of this course (UML for invisibility, requirements for conformity, tests and version control for changeability).

⚠ **Common mistake:** expecting a new framework or an AI assistant to remove the hard part. They speed up typing; they do not decide what the capacity rule should be.

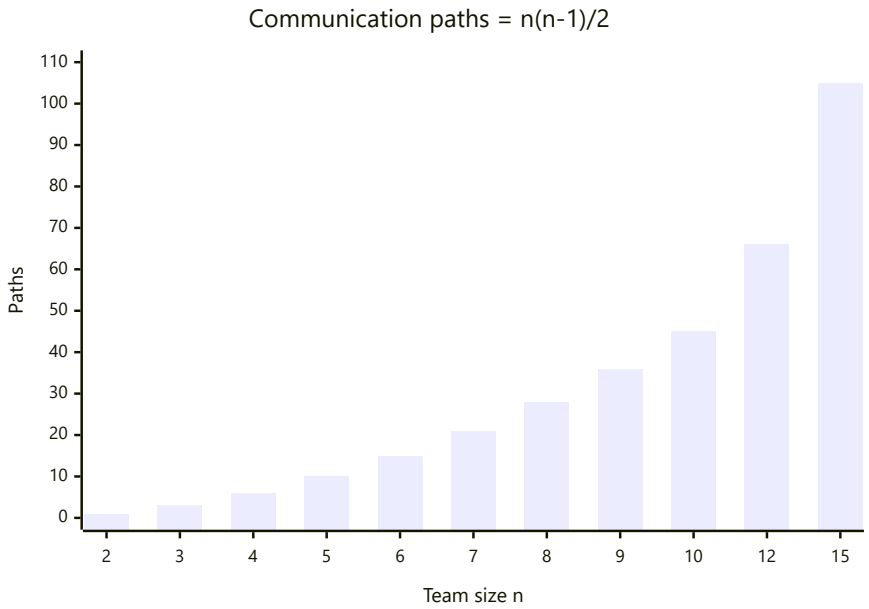
Worked example: an innocent change silently breaks a caller

```cpp
using Clock = std::chrono::system_clock;

// Week 3, Dara writes: whole HOURS until the event starts
int timeUntilStart(const Event& e, Clock::time_point now) {
    return std::chrono::duration_cast<std::chrono::hours>(
        e.start() - now).count();
}

// Week 3, Sokha uses it in RegistrationService::cancel
if (timeUntilStart(event, now) < 24) {
    throw std::runtime_error("cancellation closes 24 h before start");
}

// Week 9, Dara needs minutes for reminders and "improves" it
int timeUntilStart(const Event& e, Clock::time_point now) {
    return std::chrono::duration_cast<std::chrono::minutes>(
        e.start() - now).count();
}
// It still compiles and no test covers the 24-hour rule.
// Cancelling now closes 24 MINUTES before the start.
```

```cpp
// Fix: let the type carry the unit
std::chrono::minutes timeUntilStart(
    const Event& e, Clock::time_point now) {
    return std::chrono::duration_cast<
        std::chrono::minutes>(e.start() - now);
}

// the caller compares durations, not raw ints
using namespace std::chrono_literals;
if (timeUntilStart(event, now) < 24h) {
    throw std::runtime_error("cancellation closed");
}
```

- Both versions of `int timeUntilStart` have the same signature, so the compiler cannot warn: the **meaning** changed, not the shape.
- With `std::chrono::minutes` the comparison with `24h` converts units correctly, and passing a plain `int` no longer compiles.
- The second safety net is a unit test for the business rule "cancellation closes 24 hours before the start" (Chapter 09).

From one developer to a team: communication paths grow as n(n-1)/2



Every pair of people who may need to agree on something is one path. With *n* people there are *n*(*n*-1)/2 pairs.

```
n =  5 :  5 × 4 / 2  =    10 paths
n = 10 : 10 × 9 / 2  =    45 paths   (×4.5)
n = 50 : 50 × 49 / 2 = 1 225 paths   (×122)

Club Hub team of 5 + the client (TA) = 6
       6 × 5 / 2 = 15 paths
```

- People grow linearly, paths grow quadratically: doubling the team from 5 to 10 multiplies the paths by 4.5.
- **Brooks's law** (The Mythical Man-Month, 1975): adding people to a late project makes it later, because newcomers need training and add paths.

What changes when the team grows

Scale	Paths	How knowledge moves	Practices that become necessary
Solo (1)	0	In one head	Version control, so that "yesterday's you" can be recovered
Team (4-9)	6-36	Conversation, a shared board, pull requests	Team charter, backlog, code review, one build command, CI
Team of teams (20-100)	190-4 950	Documents, interfaces, meetings of representatives	Architecture with clear module boundaries, written requirements, release planning
Organisation (1 000+)	~500 000	Standards, platforms, internal tools	Processes, platform teams, automated compliance checks, incident management

💡 Large organisations do not let every pair talk. They **cut paths**: small teams (often 5 to 9 people) that talk to each other through well-defined interfaces, both in the code (modules, APIs) and in the organisation (one contact per team).

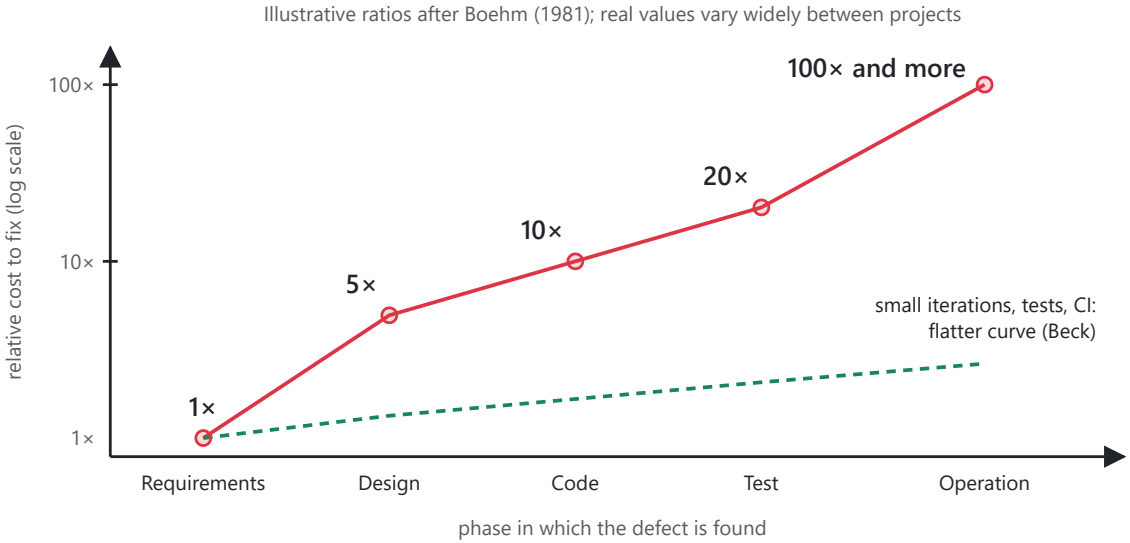
👥 Worked example: a sixth member in week 11

A Club Hub team of 5 is behind in Sprint 2 and asks for one more student.

- Paths inside the team: 10 → 15 (+50 %).
- The newcomer needs about 3 days to read the code, build it and learn the rules; someone must answer questions during those days.
- Sprint 2 ends in week 12: the help arrives when the sprint is nearly over.

Better: cut scope (move a Should feature to later), pair two existing members on the blocking task, and keep the team at 5.

The cost-of-change curve: the later a defect is found, the more it costs

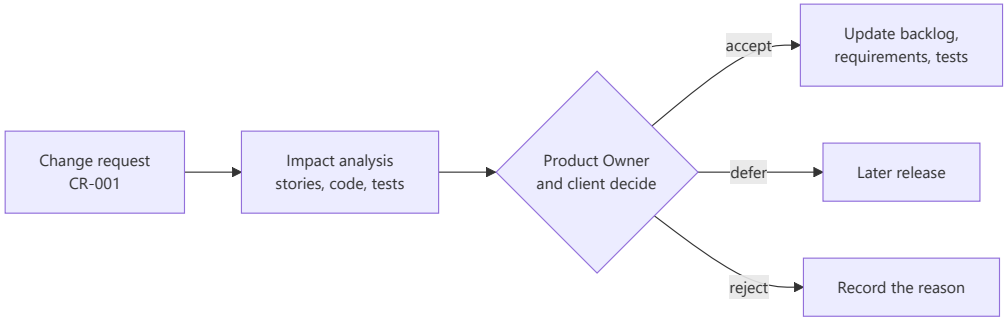


Why the curve rises: everything built on a wrong decision (design, code, tests, data, user habits) must be found and redone.

```
Club Hub, rule misunderstood:
  client means "cancel up to 24 h before"
  team writes "cancel up to 2 h before"

Found in a requirements review:
  fix one sentence                    0.5 h
Found after release:
  fix code + tests                    3.0 h
  rebuild, re-release                 1.0 h
  find and undo late cancellations    3.0 h
  apologise to 4 clubs                1.0 h
                                      -----
                                      8.0 h
Cost-of-change ratio: 8.0 / 0.5 = 16×
```

☑ Two answers: find defects early (reviews, prototypes, acceptance criteria) and keep late change cheap (small releases, automated tests, CI).

Requirements will change: decide changes instead of discovering them



Why requirements change	Club Hub example
Users see the product and learn what they want	"Can we have a waiting list?"
The environment changes	New rule: rooms above 50 people need a booking
A misunderstanding is discovered	"Capacity" meant seats, not registrations
Priorities shift	The monthly report is needed before the midterm

```
## CR-001 · Waiting list when an event is full
Requested by: Club leader, Robotics Club   Date: 2026-11-04
Description: when capacity is reached, further students join a
  waiting list; a cancellation promotes the first one waiting.
Reason: popular workshops fill in minutes; seats freed by
  cancellations stay empty.
Impact:
  - Requirements: new status RegistrationStatus::Waitlisted
  - Code: RegistrationService::registerStudent, cancel
  - Tests: 4 new cases (full, promote, cancel waitlisted, order)
  - Effort: about 6 h; Sprint 2 has 4 h of slack
Decision (PO + client, 2026-11-05): ACCEPT for Sprint 2,
  drop "export report as PDF" (Could) to make room.
```

Change is not the enemy; **unmanaged** change is. A change request makes the cost visible before anyone says yes. Chapter 05 covers requirements, Chapter 10 change impact analysis.

What users expect today: it works, it is safe from attackers, it harms no one

Reliability

Does it keep working?

- availability (uptime)
- fault tolerance, recovery
- correct results under load
- measured: % uptime, MTBF

Club Hub
registration must stay open when a popular event opens

High stakes
payments, telecom, cloud

Security

Can it resist attackers?

- confidentiality of data
- integrity: no tampering
- authenticity, accountability
- never trust input

Club Hub
student emails and phone numbers must not leak

High stakes
banking, health records

Safety

Can it harm people?

- hazards identified first
- fail-safe states
- interlocks, independent review
- domain safety standards

Club Hub
low, but a wrong capacity can overcrowd a room

High stakes
medical devices, cars, aircraft

ISO/IEC 25010:2023 lists reliability, security and (new in this edition) safety as product quality characteristics

Availability	Downtime allowed per year
99 %	3.65 days
99.9 % ("three nines")	8.76 hours
99.99 %	52.6 minutes
99.999 %	5.26 minutes

```
1 year = 365 × 24 = 8 760 h
99.9 %  ->  0.001 × 8 760 h = 8.76 h down
```

- Each extra nine costs far more than the last: redundancy, monitoring, people on call.
- Club Hub does not need five nines; it needs its *rules* to be right and its *personal data* to stay private. Choosing the right targets is Chapter 02.

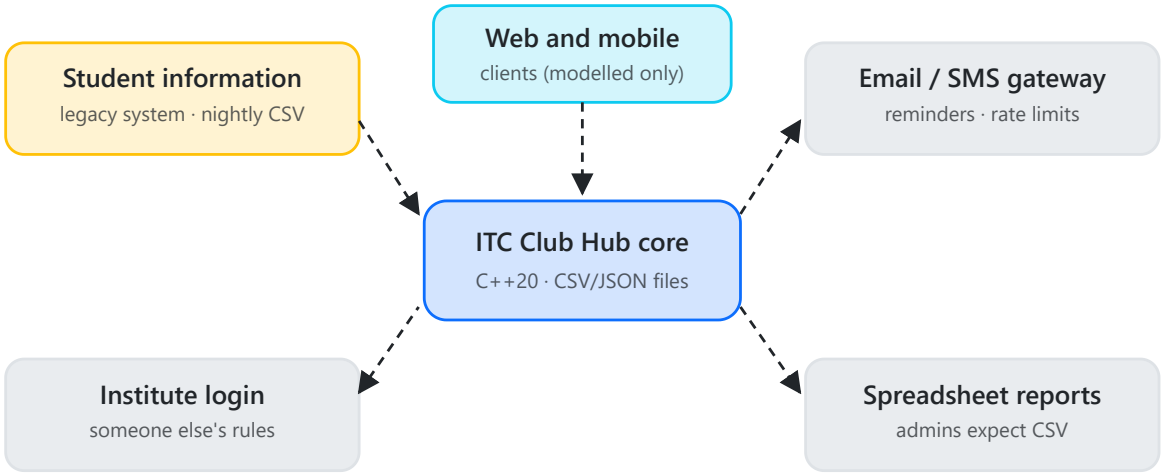
Worked example: reliable and secure code starts with input it does not trust

```cpp
#include <charconv>
#include <optional>
#include <string_view>

// The capacity column comes from events.csv, which people edit by hand.
// Reliability: never crash on bad data.
// Security: never trust input; reject anything unexpected.
std::optional<int> parseCapacity(std::string_view text) {
    int value = 0;
    const char* first = text.data();
    const char* last = text.data() + text.size();
    auto [ptr, ec] = std::from_chars(first, last, value);
    if (ec != std::errc{} || ptr != last) return std::nullopt;
    if (value < 1 || value > 500) return std::nullopt;
    return value;
}

// caller decides what "no value" means: skip the row and log it
if (auto cap = parseCapacity(fields[4])) {
    events.push_back(Event{/* ... */ *cap});
} else {
    log.warn("events.csv line {}: bad capacity", lineNo);
}
```

Input text	Result	Why
`"30"`	30	valid
`"30 "`	no value	trailing character
`"abc"`	no value	not a number
`"-5"`	no value	below 1
`"501"`	no value	above the largest hall
`"99999999999"`	no value	does not fit in `int`
`""`	no value	empty field

⚠ `std::stoi("30abc")` returns 30 and throws on `"abc"`; `std::atoi` returns 0 for garbage. Both hide bad data. `std::from_chars` reports exactly how much it parsed.

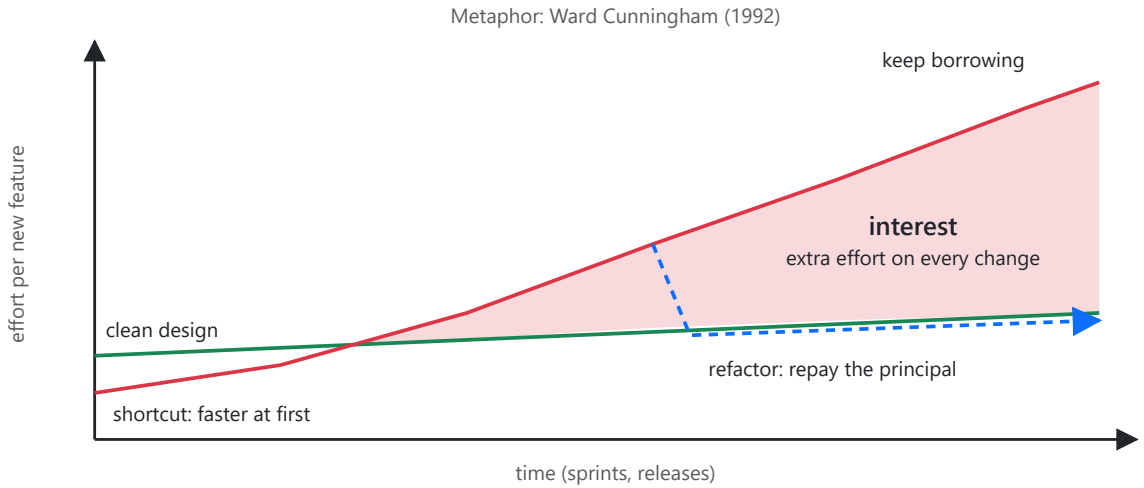
No system starts alone: legacy systems and integration



Every dashed arrow is an interface you do not fully control: agree the format, the contract and who may change it

- A **legacy system** is one that still does valuable work but is hard to change: old technology, missing documentation, original authors gone. It cannot simply be switched off.
- **Integration** is where conformity bites: the other system decides the format of a student ID, the date format in a CSV export or the number of emails per hour.
- Typical integration failures: a field changes type, a date arrives as `03/11/2026` (3 November or 11 March?), an export silently drops rows, the other side is down.
- Defence: an **adapter** per external system, validated input (previous slide), and a written interface agreement with an owner and a version.

Technical debt: a shortcut today, interest on every change tomorrow



- **Technical debt**: the future cost of a quick-and-dirty solution chosen today. The *principal* is the work needed to clean it up; the *interest* is the extra effort every later change pays while it stays.
- Debt can be a good decision (ship the demo on Friday) if it is **deliberate, recorded and repaid**. Unnoticed debt is the dangerous kind.
- Forms: duplicated code, missing tests, hard-coded values, outdated dependencies, no documentation, "temporary" workarounds.
- Legacy systems are often systems whose debt was never repaid.

☑ Record debt where the team sees it: a GitHub issue labelled `tech-debt` with the principal (estimate) and the interest (what it slows down).

Worked example: copy-pasted validation becomes debt

```
// Sprint 1: quick and it works
void RegistrationService::registerStudent(const Student& s,
                                          const Event& e) {
    if (s.email().find('@') == std::string::npos) {
        throw std::invalid_argument("invalid email");
    }
    // ... register the student for e
}

// Sprint 2: copy, paste, adjust a little
void ClubService::addMember(const Student& s, Club& c) {
    if (s.email().find('@') == std::string::npos ||
        s.email().size() > 100) {
        throw std::invalid_argument("bad email");
    }
    // ... add s to c
}
// The two rules have already drifted apart: only one checks length.
// Sprint 3: "accept institute addresses only" must be found and
// changed in every copy. Miss one and the rule is half enforced.
```

```
// Repay: one rule, one place, one test
namespace clubhub {

[[nodiscard]] bool isValidEmail(std::string_view email) {
    const auto at = email.find('@');
    return at != std::string_view::npos && at > 0
        && at + 1 < email.size() && email.size() <= 100;
}

void requireValidEmail(std::string_view email) {
    if (!isValidEmail(email)) {
        throw std::invalid_argument("invalid email");
    }
}

}  // namespace clubhub
// both services now call requireValidEmail(s.email())
```

Per rule change	2 copies	1 function
Places to edit	2 (if you find them)	1
Tests to update	2 sets, often none	1 set
Risk of a half-applied rule	high	none

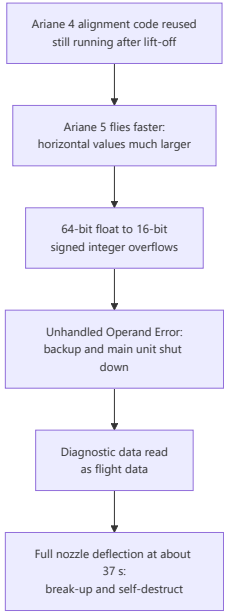
Four failures, four lessons

Case	What happened	Root cause	Essential difficulty	Practice that would have helped
Therac-25 1985-1987 radiation therapy	At least six patients received massive radiation overdoses; several died. The machine showed a cryptic "Malfunction" message and operators could simply continue.	A race condition when an operator edited the treatment quickly; hardware interlocks of earlier models had been removed and their job left to software reused from those models.	Complexity, invisibility	Independent safety analysis, hardware interlocks for hazards, concurrency testing, incident reporting and investigation (Chapters 03, 09).
Ariane 5 flight 501, 1996 rocket	About 37 seconds after lift-off the launcher veered off course and self-destructed on its maiden flight.	Inertial reference software reused from Ariane 4 converted a 64-bit float to a 16-bit integer; Ariane 5's larger values overflowed and both units shut down.	Conformity, changeability	Re-validate reused code against the new requirements, test with realistic flight data, handle conversion errors (Chapters 05, 09).
Knight Capital 1 August 2012 stock trading	In about 45 minutes the firm's trading system sent millions of unintended orders and lost about 440 million US dollars.	New code was deployed manually to 8 servers and missed one; a reused feature flag woke up old, dead code on that server.	Changeability, invisibility	Automated, verified deployment, removing dead code, never reusing flags, a tested kill switch (Chapters 08, 10).
CrowdStrike 19 July 2024 security software	A faulty content update to a Windows security agent crashed about 8.5 million computers; airlines, hospitals and banks stopped for hours to days.	A configuration file with an unexpected number of fields caused an out-of-bounds memory read in a kernel driver; the update went to all customers at once.	Complexity, conformity	Validate every input (even your own config), staged rollouts to a small group first, automatic rollback (Chapters 08, 09).

💡 The pattern repeats: **reused or changed code** meets **a situation nobody re-checked**, and **nothing stops the damage** from spreading.

📖 Facts as widely reported by the official inquiry (Ariane 5), the regulator (Knight Capital: US SEC) and the vendor's own review (CrowdStrike); Therac-25 after Leveson and Turner (1993).

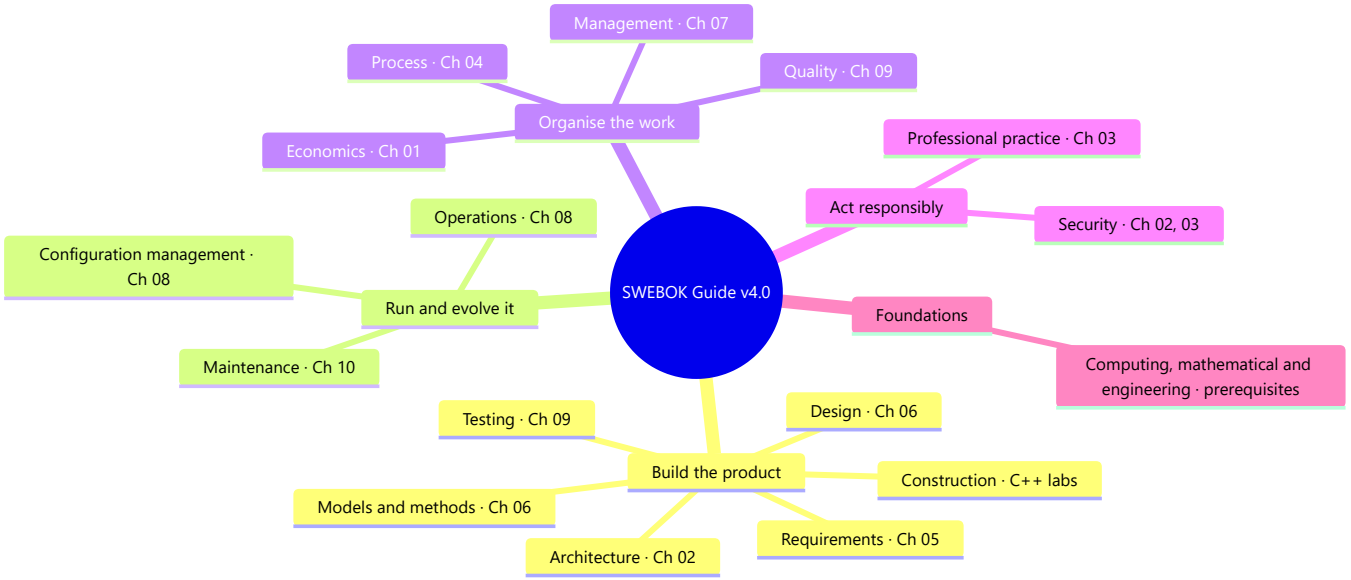
Ariane 5: from reused code to explosion, and the same bug in C++




```cpp
#include <cstdint>
#include <limits>
#include <optional>

// Ariane 4 assumption: the horizontal bias always fits in 16 bits
std::int16_t toBias(double horizontalBias) {
    // unchecked narrowing: undefined behaviour when out of range
    return static_cast<std::int16_t>(horizontalBias);
}
// toBias(40000.0): 40000 is outside [-32768, 32767].
// One MSVC x64 build printed -25536; another compiler may do anything.

// Checked version: the caller must decide what to do
std::optional<std::int16_t> toBiasChecked(double v) {
    using L = std::numeric_limits<std::int16_t>;
    if (!(v >= L::min() && v <= L::max())) {
        return std::nullopt;   // also rejects NaN
    }
    return static_cast<std::int16_t>(v);
}
```

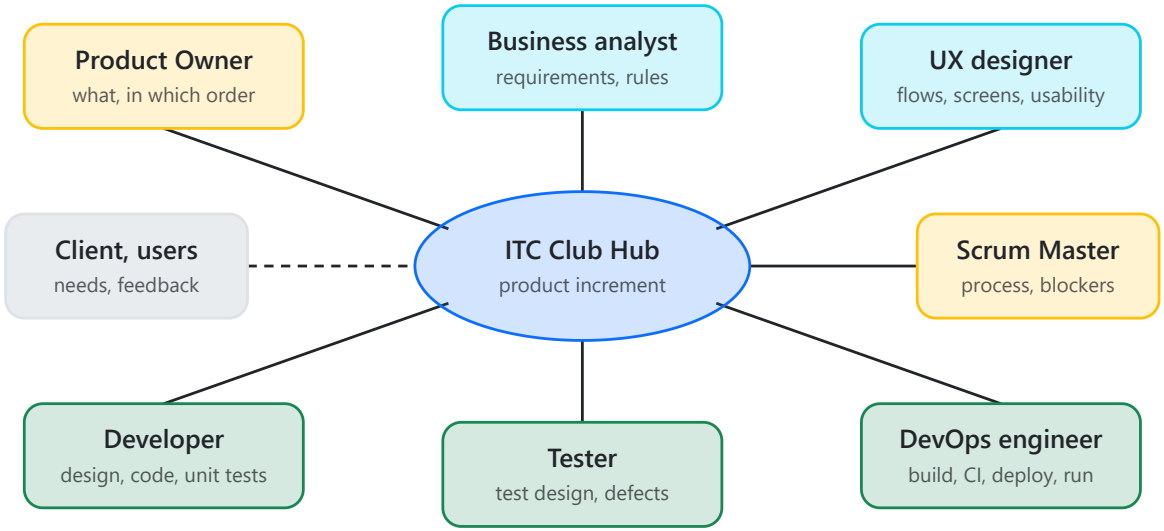
⚠ The Ada code on Ariane did detect the overflow and raised an exception, but nobody handled it, because the analysis for Ariane 4 had shown the value could never be that large. The requirement changed; the code and its assumptions did not. (Ariane 501 Inquiry Board report, 1996)

SWEBOK Guide v4.0: a map of the discipline and of this course



- The **Software Engineering Body of Knowledge** (IEEE Computer Society) describes what a software engineer with a few years of experience is expected to know.
- **Version 4.0 (2024)** has 18 knowledge areas: 15 engineering areas plus 3 foundations. New since v3: Software Architecture, Software Engineering Operations and Software Security.
- This course touches every engineering area once, at survey level. The Software Engineering and IT Project Management courses go deeper into construction, design, management and economics.
- Use the map when a topic feels isolated: it tells you where it belongs and which chapter treats it.

Roles in a software team: who does what around the product



Role	Main output	Ch.
Product Owner	Ordered product backlog, acceptance decisions	07
Business analyst	Requirements, business rules, SRS	05
UX designer	User flows, wireframes, usability checks	02
Developer	Design, code, unit tests, reviews	06
Tester	Test cases, defect reports, quality view	09
DevOps engineer	Build, CI/CD pipeline, monitoring	08

A role is a **responsibility**, not a person. In a Club Hub team of 5, everyone develops, one student is Product Owner, the Scrum Master rotates each sprint, and the other roles are shared hats.

Worked example: a team charter makes roles and agreements explicit

```
# Team charter · Angkor Coders (v1.0, 2026-10-02)

## Members and roles
| Member     | Role (Sprint 1) | Also covers          | GitHub      |
|------------|-----------------|----------------------|-------------|
| Chan Dara  | Product Owner   | business analysis    | @chandara   |
| Sok Sokha  | Scrum Master    | testing              | @soksokha   |
| Heng Vanna | Developer       | DevOps: build, CI    | @hvanna     |
| Keo Malis  | Developer       | UX sketches          | @keomalis   |
| Lim Piseth | Developer       | testing              | @lpiseth    |

## Working agreements
- Stand-up Mon/Wed/Fri 12:30, 10 minutes, same channel.
- Every change goes through a pull request with one review.
- Done = it builds, tests pass, README updated.

## Communication channels
- Chat: team group. Work items: GitHub Issues. Files: the repo.

## Decision rules
- Product questions: the Product Owner decides after listening.
- Technical questions: consensus in 24 h, else majority vote.
- Every decision goes into lab01/decisions.md with a date.
```

- A **team charter** is a one-page agreement written by the team for the team: who is in it, who plays which role, how it works, talks and decides.
- It answers the questions that otherwise cause conflict in week 10: "who was supposed to review this?", "why did you merge without asking?", "who decided that?".
- Decision rules matter most: without them a team of 5 either argues forever or lets the loudest person decide.
- It is a living document: revise it after each sprint retrospective and bump the version.

Lab-01 Task 1: your team writes `lab01/team-charter.md` with exactly these sections.

Best practices and common mistakes

1 Version control from day one

Every change, by every member, in Git with a clear message. It manages changeability and makes individual work visible.

2 Find defects where they are cheap

Review requirements and designs before coding; write acceptance criteria. A sentence fixed in week 5 saves a release in week 12.

3 Let types and tests guard the rules

Units in `std::chrono` types, checked conversions, `std::optional` for "no value", and one unit test per business rule.

4 Keep teams small, interfaces clear

Cut communication paths with modules, a written charter and one contact per external party (the Product Owner for the client).

5 Borrow technical debt on purpose

Record every shortcut as a `tech-debt` issue and repay a little each sprint, before the interest dominates.

6 Re-check assumptions when reusing

Reused code carries the assumptions of its old context (Ariane 5, Therac-25). Re-validate it against the new requirements and data.

⊗ **Common mistakes:** coding before the problem is understood · "it works on my machine" as the test · one member doing all commits · adding people late instead of cutting scope · treating a new tool as a silver bullet · deploying by hand to "just one more server".

Chapter quiz 10 questions

1. According to Brooks, turning a program into a programming systems product costs roughly how much more effort?

- The same effort
- About twice
- About three times
- About nine times

2. Which of these is an accidental difficulty rather than one of Brooks's four essential difficulties?

- Complexity
- Conformity
- Slow compilers and poor tools
- Invisibility

3. Where was the term "software engineering" popularised as an answer to the software crisis?

- At the 1968 NATO conference in Garmisch
- In the Agile Manifesto of 2001
- In the first SWEBOK Guide
- In the Ariane 5 inquiry report

4. A team grows from 5 to 10 members. Using n(n-1)/2, how do the communication paths change?

- From 10 to 20
- From 10 to 45
- From 25 to 100
- From 5 to 10

5. int timeUntilStart(...) used to return hours; a teammate changes it to return minutes, and the caller still checks "< 24". Which change lets the compiler protect the caller?

- Add a comment saying the unit is now minutes
- Return std::chrono::minutes and compare with 24h
- Rename the variable in the caller
- Return a double instead of an int

6. Why is it cheaper to find a misunderstood requirement in a review than after release?

- Reviews need no people
- After release, design, code, tests, data and user habits built on it must be found and redone
- Requirements may not change after coding starts
- Released software cannot be changed at all

7. A service promises 99.9 % availability. About how much downtime per year does that allow?

- 52.6 minutes
- 8.76 hours
- 3.65 days
- 87.6 hours

8. The same email check is copy-pasted into registerStudent and addMember. In the technical debt metaphor, what is the interest?

- The time it took to write the first copy
- The extra effort and risk on every later rule change, which must be made in every copy
- The licence cost of the compiler
- The memory used by the duplicated code

9. What does static_cast<std::int16_t>(40000.0) have in common with the Ariane 5 failure?

- Nothing, 40000 fits in 16 bits
- The value is outside [-32768, 32767], so the conversion overflows, like the horizontal bias did
- It is slow, and Ariane 5 failed because of timing
- It only drops the fractional part, which is harmless

10. In an ITC Club Hub team, who represents the client and decides the order of the product backlog?

- The Scrum Master
- The Product Owner
- The tester
- The DevOps engineer

WRAP-UP

Summary

- Software engineering is a systematic, disciplined, quantifiable approach to developing, operating and maintaining software.
- A product costs about 9 times a program: it must be general, tested, documented and fit into a larger system.
- The term was popularised at the 1968 NATO conference as the answer to the software crisis.
- Brooks's essential difficulties (complexity, conformity, changeability, invisibility) can be managed, never removed.
- Communication paths grow as n(n-1)/2: 10 for 5 people, 45 for 10, 1 225 for 50.
- The later a defect is found, the more it costs; find it early and keep late change cheap.
- Reliability, security and safety are expectations, not extras; never trust input.
- Technical debt is borrowed time: record it, and repay it before the interest dominates.
- Failures (Therac-25, Ariane 5, Knight Capital, CrowdStrike) come from unchecked assumptions; roles and SWEBOK show who guards against them.

Open Lab-01: 5 tasks + 1 challenge

Next chapter: 02 · Types of Applications