INTRODUCTION TO SOFTWARE ENGINEERING · CHAPTER 10

Software Maintenance and Evolution

Most of the cost of software comes after its first release. This chapter covers how systems change over time and how to keep them healthy: maintenance categories, technical debt, refactoring, versioning, change control and release management.

ISO/IEC/IEEE 14764:2022 | Lehman's laws · SemVer 2.0.0 · Keep a Changelog | C++20 · CMake · GoogleTest

AGENDA

What we will cover

1. Types of maintenance: corrective, adaptive, perfective, preventive (ISO/IEC/IEEE 14764)
2. Lehman's laws of software evolution
3. Legacy systems and modernisation options
4. Technical debt: causes, measurement and repayment
5. Code smells and refactoring
6. Reverse engineering and documentation recovery
7. Configuration management, versioning and semantic versioning
8. Change requests and change impact analysis
9. Release management, changelogs and deprecation
10. Maintainability metrics and maintenance planning
11. Chapter Quiz
12. Lab-10

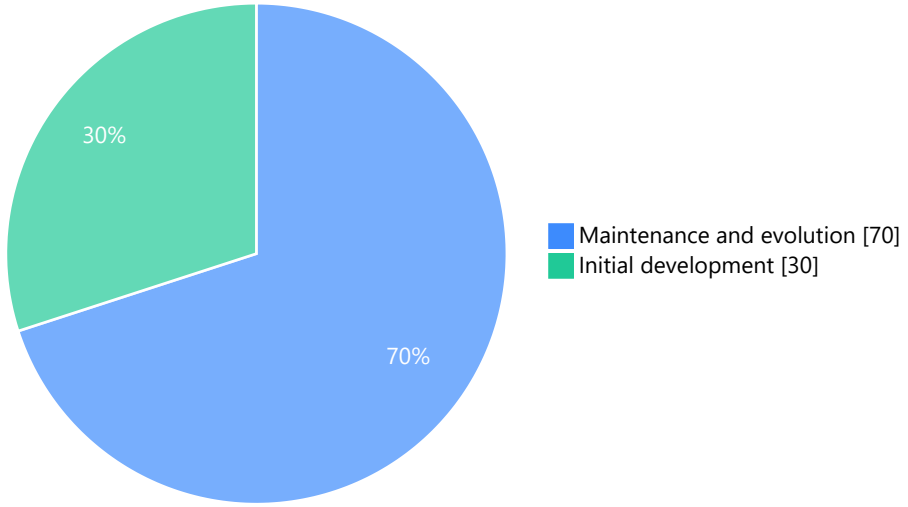
Click any item to jump directly to that topic.

Software is delivered once and maintained for years

- **Software maintenance**: every modification of a software product *after delivery*, to fix faults, adapt it to a new environment, or improve it (SWEBOK Guide v4.0, Software Maintenance knowledge area).
- **Software evolution**: the long view of the same work. How a system grows, changes shape and ages over many releases.
- Studies since the 1970s report that maintenance takes most of the life-cycle cost, typically quoted as **60 to 80 %**, sometimes more. The exact share depends on the system and on what is counted, so treat any single number with care.
- **ITC Club Hub**: your team builds it in weeks 9 to 14. If the clubs office adopts it, next year's students will maintain code they did not write. Everything in this chapter is about making that possible.

> 💡 **Cost-of-change link (Chapter 01):** the later a change is made, the more it costs. Maintenance is where every shortcut taken during development is finally paid for.

Life-cycle cost, typical split (illustrative)



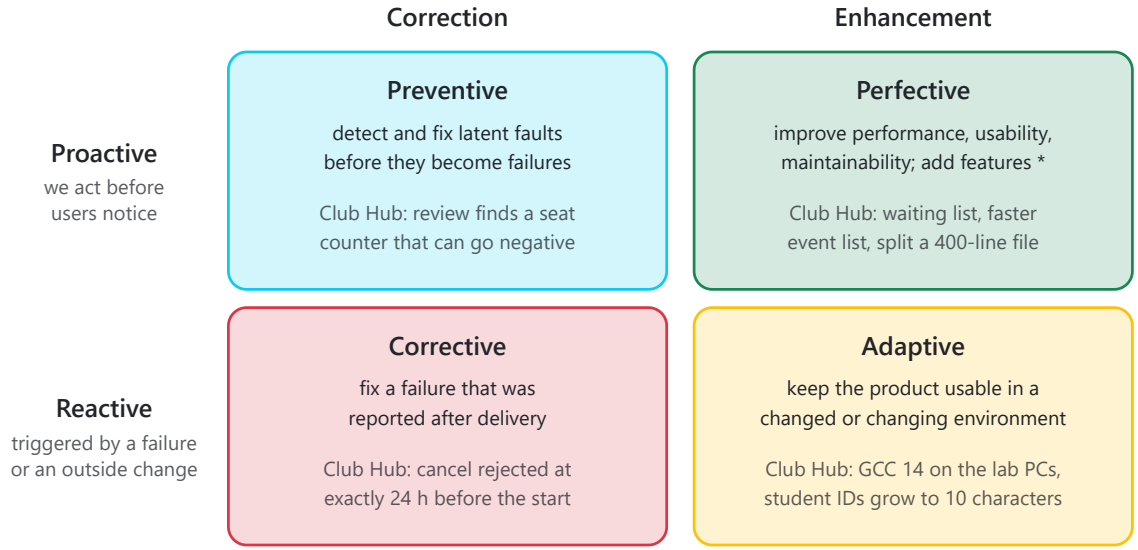
Illustrative split at the middle of the 60 to 80 % range reported in the literature; not a measurement of a real project.

Four categories, sorted by two questions

- ISO/IEC/IEEE 14764:2022 classifies maintenance by **why** a change is made, not by how big it is.
- **Question 1**: is something wrong (*correction*), or is something new or better needed (*enhancement*)?
- **Question 2**: was it triggered by a failure or an outside change (*reactive*), or did we act before anyone noticed (*proactive*)?
- The category drives priority and version: corrective work is often urgent and becomes a PATCH; perfective work is planned in the backlog and usually becomes a MINOR release.

Where the effort goes (classic survey)	Share
Perfective (new features, improvements)	about 50 %
Adaptive (environment changes)	about 25 %
Corrective (bug fixing)	about 21 %
Other	about 4 %

Lientz and Swanson (1980). Bug fixing is only about one fifth of maintenance.



ISO/IEC/IEEE 14764:2022 · * the 2022 edition also names new features "additive" maintenance

Worked example: ten Club Hub change requests, classified

CR	Request	Trigger	Category
01	The CLI crashes when `events.csv` has an empty description field	user report	Corrective
02	Cancelling exactly 24 h before the start is rejected, although the rule allows it	user report	Corrective
03	Code review: the seat counter can go negative if `cancel` is called twice; no user has hit it yet	found by us	Preventive
04	clang-tidy reports a use-after-move in the reminder code; tests still pass	found by us	Preventive
05	Lab PCs move to GCC 14; the build fails on a missing `#include <algorithm>`	environment	Adaptive
06	The institute changes student IDs from 9 to 10 characters	environment	Adaptive
07	The IT office requires JSON instead of CSV for all data files	environment	Adaptive
08	Add a waiting list for full events (a Should feature)	users want more	Perfective
09	Export the attendance of an event as CSV	users want more	Perfective
10	Split the 400-line `Cli.cpp` into small command handlers	maintainers	Perfective

- **Tally**: 2 corrective, 2 preventive, 3 adaptive, 3 perfective.
- CR-02 and CR-03 are both faults. CR-02 was *reported* (reactive, corrective); CR-03 was found *before* any failure (proactive, preventive).
- CR-06 feels like a bug to the users, but the software did not change: the environment did. That makes it adaptive.
- CR-08 and CR-09 add functionality; the 2022 edition of the standard calls this *additive*, a kind of perfective enhancement.

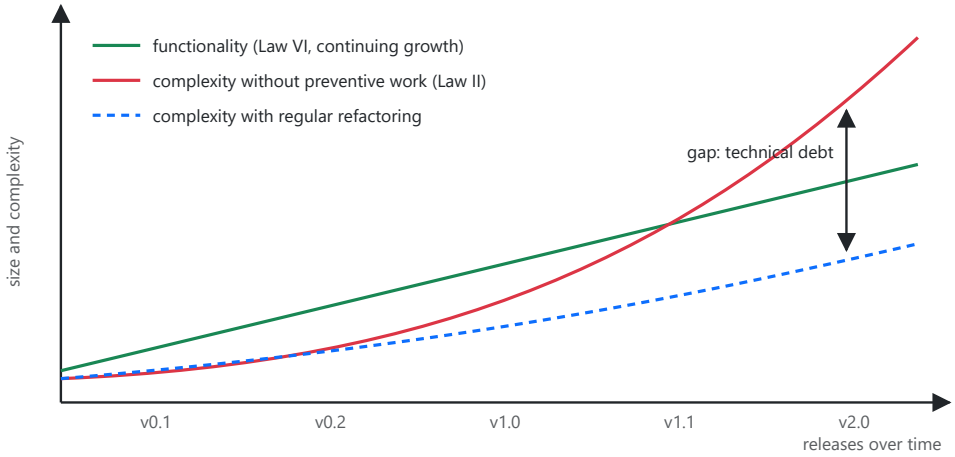
⚠ **Classify by the reason, not by the code.** The same edit in `RegistrationService.cpp` can be corrective in one request and perfective in another.

Many textbooks call refactoring (CR-10) *preventive*. ISO/IEC/IEEE 14764 files "improve maintainability" under perfective and keeps preventive for latent faults. Say which convention your team uses.

Lehman's laws: what happens to software that is actually used

#	Law (year)	In plain words
I	Continuing change (1974)	A used system must keep changing, or it becomes less and less useful.
II	Increasing complexity (1974)	As it changes, its complexity grows unless work is done to reduce it.
III	Self-regulation (1974)	Growth follows regular, predictable trends over releases.
IV	Conservation of organisational stability (1978)	The effective work rate stays about constant; more people barely speed it up.
V	Conservation of familiarity (1978)	Each release can only add so much; a release with too much new content causes trouble.
VI	Continuing growth (1991)	Functionality must keep growing to keep users satisfied.
VII	Declining quality (1996)	Quality seems to decline unless the system is actively adapted.
VIII	Feedback system (1996)	Evolution is a feedback process: users, market and team react to each release.

Lehman (1974 to 1996), for **E-type** systems: programs embedded in the real world, like Club Hub. An **S-type** program is fully defined by a specification (a sort function) and does not evolve this way.



For Club Hub: Law I says the waiting list and CSV export will come; Law II says each one makes the code harder to change unless the team spends time on refactoring. That is why every sprint from now on reserves time for it.

Legacy systems: old, valuable and hard to change

- A **legacy system** is an older system that the organisation still depends on, but that is costly and risky to change.
- Typical signs: outdated language or platform, the original developers have left, little or no documentation, **no automated tests**, business rules that exist only in the code, data in formats nobody else reads.
- Michael Feathers (*Working Effectively with Legacy Code*, 2004) gives a practical definition: **legacy code is code without tests**. By that definition a two-month-old student project can already be legacy.
- Why not simply throw it away? It works, it encodes years of decisions, and a rewrite must reproduce all of that before it delivers any value.

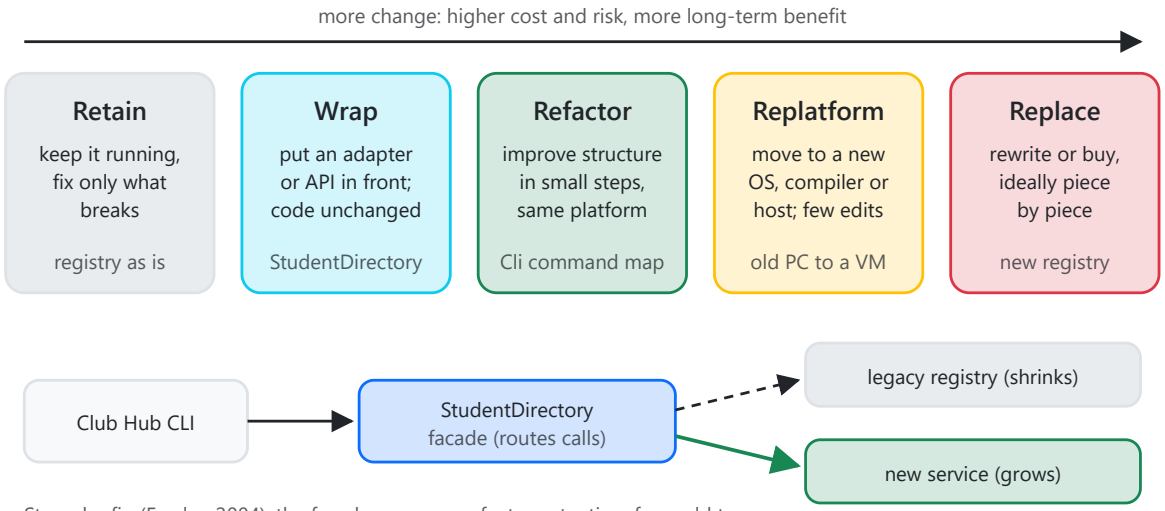
🏢 **Scenario:** the institute's student registry was written in 2009 as a desktop application with its own database. One staff member understands it. Club Hub needs student names and emails from it. What do we do with the registry?

First assess, then decide: rate each legacy system on business value and technical quality (after Sommerville, *Software Engineering*).

	Low technical quality	**High technical quality**
High business value	**Modernise**: wrap, refactor, replatform or replace step by step. *The student registry.*	**Keep maintaining** it normally.
Low business value	**Retire** it; move the few users elsewhere.	**Retain** with minimal effort, or replace with an off-the-shelf product when convenient.

Assessment question	**Student registry**
How many people and processes depend on it?	whole institute: high value
Can we build it and run tests today?	builds only on one old PC, no tests
Who understands it?	one person: bus factor 1
Is the platform still supported?	no security updates since 2020

Five modernisation options, from least to most change



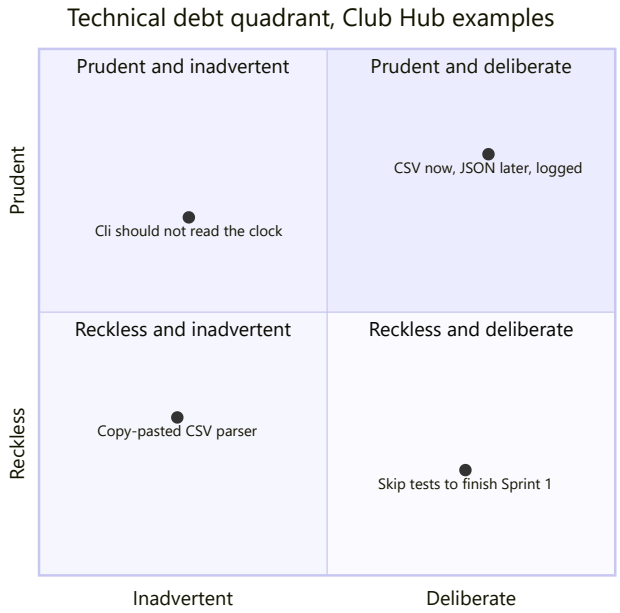
Strangler fig (Fowler, 2004): the facade moves one feature at a time from old to new

Choose	when
Retain	it works, changes are rare, and it will retire soon
Wrap	new systems need its data or functions, but its inside is too risky to touch
Refactor	the platform is fine, but the code is hard to change; tests can be added
Replatform	the code is acceptable, but the hardware, OS or compiler is dying
Replace	value is high, quality is low, and the rules are well understood

Decision for the registry: *wrap* now (Club Hub reads students only through a `StudentDirectory` interface), and plan a *replace* later with the strangler fig, one feature at a time.

Big-bang rewrites are the riskiest option: nothing ships until everything is reproduced, and the old system keeps changing meanwhile.

Technical debt: shortcuts that charge interest



Quadrant after Martin Fowler (2009); the metaphor is Ward Cunningham's (1992).

- **Technical debt**: the future cost of choosing an easy solution now instead of a better one. Like money debt it has a **principal** (effort to fix it properly) and **interest** (extra effort every change costs while it stays).
- **Deliberate** debt is a conscious trade-off; **inadvertent** debt is discovered later ("now we know how we should have done it"). **Prudent** debt is recorded with a plan; **reckless** debt is not.
- Only prudent, deliberate debt is a healthy engineering decision. Inadvertent prudent debt is normal learning; reckless debt is the one to stop.

Common causes	Kinds of debt
deadline pressure (the sprint demo)	code: smells, duplication
missing knowledge or experience	design: wrong responsibilities
requirements that changed after design	test: missing or weak tests
no time planned for refactoring	documentation: outdated diagrams
dependencies never updated	build: old compiler, pinned libraries

Worked example: a technical-debt register with interest estimates

Id	Debt item (file)	Principal	Interest per sprint	Interest / principal	Decision
TD-01	41-line if-else chain in `Cli::handleCommand`, CC 11 (`src/Cli.cpp`)	6 h	2 h (every new command edits it)	0.33	repay in Sprint 2
TD-02	magic number `24` in three places	1 h	0.5 h + risk of an inconsistent rule	0.50	repay now
TD-03	CSV splitting copied into two repositories	4 h	1 h (fix bugs twice)	0.25	repay in Sprint 2
TD-04	no tests for the reminder scheduler	5 h	1 h + high risk	0.20	next sprint
TD-05	hard-coded path `"data/events.csv"`	2 h	0.2 h	0.10	accept, revisit

Rule of thumb: repay the items with the highest interest-to-principal ratio first, and items in code you are about to change anyway.

🧮 **Measuring: technical debt ratio**

debt ratio = remediation cost ÷ development cost

Club Hub: principal 6 + 1 + 4 + 5 + 2 = **18 h**; development so far 5 students × 60 h = **300 h**.

18 ÷ 300 = **6 %**. SonarQube's default scale rates up to 5 % as A and 6 to 10 % as B.

- **Make it visible**: one register row per item, linked from a `// TD-03` comment in the code and a backlog item.
- **Pay continuously**: reserve a fixed share of each sprint (for example 15 to 20 %) instead of a "refactoring sprint" that never comes.
- **Boy Scout rule**: leave every file a little cleaner than you found it.

⚠ Interest is an *estimate*. Its value is in the ranking and the conversation with the Product Owner, not in the decimals.

Code smells: surface symptoms of deeper design problems

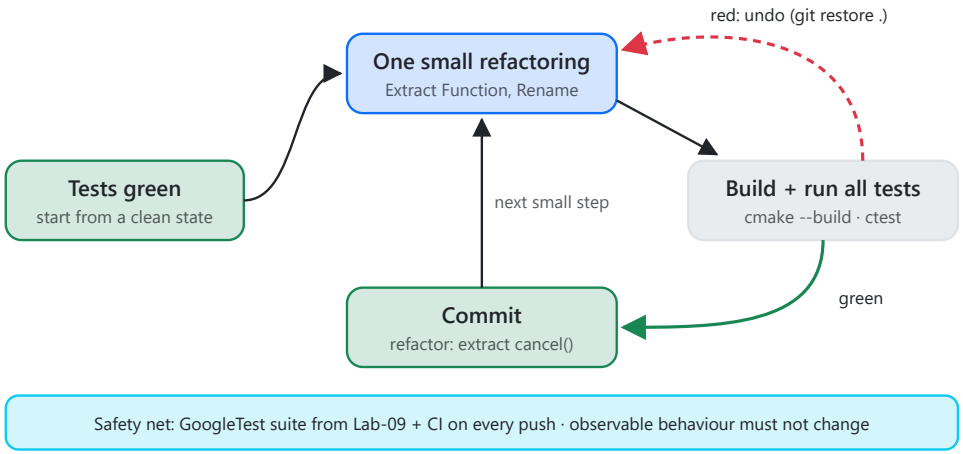
Smell	What it looks like in Club Hub	Typical refactoring
Long Function	`Cli::handleCommand`: 41 lines, five branches	Extract Function
Duplicated Code	the same usage check in three branches; CSV splitting in two repositories	Extract Function, then call it
Repeated Switches	if-else chains on command strings or status values in several places	Replace Conditional with Polymorphism, or a lookup map
Mysterious Name	`ok()`, `t`, `n2`, `tmp`	Rename Function, Rename Variable
Magic numbers	`24`, `3600`, `86400` scattered in the code	Replace Magic Literal with a named constant
Long Parameter List	`createEvent(title, date, time, room, cap, desc, clubId)`	Introduce Parameter Object (`EventDraft`)
Primitive Obsession	student ID as a raw `std::string`, validated in five places	Replace Primitive with Object (`StudentId`)
Feature Envy	`Cli` computes whether an event is full from its fields	Move Function (to `Event::isFull()`)
Shotgun Surgery	adding a registration status touches six files	Move Function, Combine Functions into Class
Large Class	`RegistrationService` also parses CSV and prints output	Extract Class
Global Data	globals such as `g_events` in old report code	Encapsulate Variable, pass as parameter

Names from Fowler, *Refactoring*, 2nd ed. (2018), except "magic numbers", a common name for a special case of Mysterious Name.

- A **code smell** is not a bug: the program works. It is a hint that the code will be expensive to change.
- Smells are judged in context: a 50-line function that reads like a recipe may be fine.
- Tools find many smells for you. The `.clang-tidy` file from Lab-09 can flag long and complex functions and magic numbers:

```
Checks: >
  readability-function-size,
  readability-function-cognitive-complexity,
  readability-magic-numbers
CheckOptions:
  - key: readability-function-size.LineThreshold
    value: 40
  - key: readability-function-cognitive-complexity.Threshold
    value: 15
```

Refactoring: small steps behind a safety net of tests



Refactoring: a change to the internal structure of software that makes it easier to understand and cheaper to modify *without changing its observable behaviour* (Fowler, *Refactoring*, 2018). Kent Beck's **two hats**: either you add behaviour or you refactor, never both in one commit.

```cpp
// tests/CliTest.cpp: pins the CLI behaviour BEFORE refactoring
class CliTest : public ::testing::Test {
protected:
    // in-memory fakes from Lab-09
    InMemoryEventRepository events;
    InMemoryRegistrationRepository registrations;
    RegistrationService service{events, registrations};
    std::ostringstream out;
    Cli cli{service, events, out};
};

TEST_F(CliTest, UnknownCommandIsReported) {
    cli.handleCommand("dance");
    EXPECT_EQ(out.str(), "unknown command: dance\n");
}

TEST_F(CliTest, CancelWithoutEventIdPrintsUsage) {
    cli.handleCommand("cancel e20230001");
    EXPECT_EQ(out.str(), "usage: cancel <student> <event>\n");
}

TEST_F(CliTest, EmptyLineIsAnUnknownCommand) {
    cli.handleCommand("");
    EXPECT_EQ(out.str(), "unknown command: \n");
}
```

The third test pins an odd case on purpose: if the refactored code silently ignores empty lines, that is a behaviour change, and the test says so.

Three everyday refactorings: rename, extract function, name the constant

```cpp
// BEFORE: what is ok? what is t? why 24?
bool ok(const Event& e, TimePoint t) {
    return e.start - t >= std::chrono::hours{24};
}
// ...and in printReport(), the same expression three times:
double r = reg == 0 ? 0.0 : 100.0 * in / reg;

// AFTER 1: Rename Function, Rename Variable,
//          Replace Magic Literal with a named constant
constexpr auto kCancellationWindow = std::chrono::hours{24};

bool isCancellationOpen(const Event& event, TimePoint now) {
    return event.start - now >= kCancellationWindow;
}

// AFTER 2: Extract Function (one definition, three callers,
//          and now it can have its own unit test)
double attendanceRate(int checkedIn, int registered) {
    if (registered == 0) return 0.0;
    return 100.0 * checkedIn / registered;
}
```

Refactoring	Mechanics in short
Rename	use the IDE's rename (it updates every caller); build; test
Replace Magic Literal	add a `constexpr` constant with a unit-carrying type (`std::chrono::hours`); replace each literal; test
Extract Function	copy the fragment into a new function; pass what it reads as parameters; replace the fragment with a call; test

```
# one refactoring = one commit, tests green each time
git commit -m "refactor: rename ok() to isCancellationOpen()"
git commit -m "refactor: add kCancellationWindow constant"
git commit -m "refactor: extract attendanceRate()"
```

⚠ Refactoring preserves behaviour, even wrong behaviour. If you notice a bug while refactoring, write it down, finish the refactoring, then fix the bug test-first in its own commit.

Before: one function knows every command

```
// src/Cli.cpp (before)
void Cli::handleCommand(const std::string& line) {
    std::istringstream in{line};
    std::string cmd, sid;
    int eid = 0;
    in >> cmd;
    if (cmd == "list") {
        for (const Event& e : events_.findAll()) {
            out_ << e.id << ' ' << e.title << '\n';
        }
    } else if (cmd == "register") {
        if (!(in >> sid >> eid)) {
            out_ << "usage: register <student> <event>\n";
            return;
        }
        service_.registerStudent(sid, eid);
        out_ << "registered\n";
    } else if (cmd == "cancel") {
        if (!(in >> sid >> eid)) {
            out_ << "usage: cancel <student> <event>\n";
            return;
        }
        const Event e = events_.findById(eid);
        if (e.start - Clock::now() < std::chrono::hours{24}) {
            out_ << "too late to cancel\n";
            return;
        }
        service_.cancel(sid, eid);
        out_ << "cancelled\n";
    } else if (cmd == "checkin") {
        if (!(in >> sid >> eid)) {
            out_ << "usage: checkin <student> <event>\n";
            return;
        }
        service_.checkIn(sid, eid);
        out_ << "checked in\n";
    } else if (cmd == "help") {
        out_ << "list | register | cancel | checkin | help\n";
    } else {
        out_ << "unknown command: " << cmd << '\n';
    }
}
```

Smell found	Evidence
Long Function	41 lines, cyclomatic complexity 11
Repeated Switches, Divergent Change	every new command edits this function
Duplicated Code	three identical "read student and event, else usage" blocks
Magic number	`24` hours, also used elsewhere
Feature Envy	a business rule (the 24 h window) lives in the user interface

Plan: five small steps, one commit each

1. Pin behaviour: Lab-09 tests + the new `CliTest` cases; all green.
2. Extract each branch into a member function (`list`, `registerStudent`, `cancel`, `checkIn`, `help`).
3. Extract the duplicated parsing into `studentAndEvent()`.
4. Replace the if-else chain with a command map.
5. Move the 24 h rule behind `isCancellationOpen()` and `kCancellationWindow`.

💡 Build and run `ctest` after every step. If a step turns red and the reason is not obvious in two minutes, `git restore .` and take a smaller step.

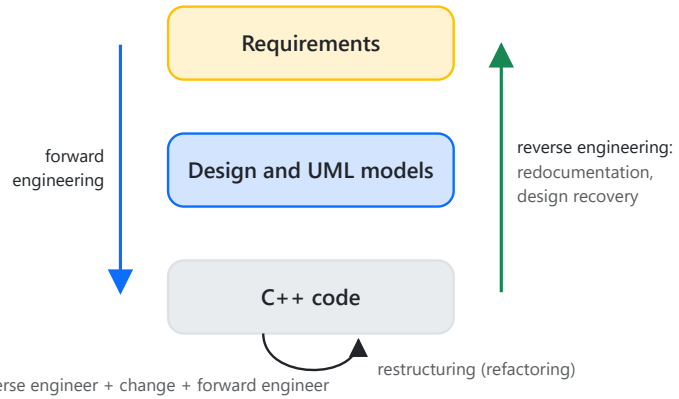
After: small functions and a command map, same behaviour

```
// include/clubhub/Cli.hpp (after)
using Args = std::vector<std::string>;
using Handler = std::function<void(const Args&)>;

struct StudentEvent {
    std::string studentId;
    int eventId = 0;
};

class Cli {
public:
    Cli(RegistrationService& service, const EventRepository& events,
        std::ostream& out);
    void handleCommand(const std::string& line);

private:
    void list(const Args& args);
    void registerStudent(const Args& args);
    void cancel(const Args& args);
    void checkIn(const Args& args);
    void help(const Args& args);
    std::optional<StudentEvent> studentAndEvent(const Args& args,
                                                std::string_view usage);

    RegistrationService& service_;
    const EventRepository& events_;
    std::ostream& out_;
    std::unordered_map<std::string, Handler> commands_;
};
```

☑ Adding a command now means one new member function and one map entry; `handleCommand` never changes again. All Lab-09 tests and the new `CliTest` cases stay green, so the behaviour is unchanged.

```
// src/Cli.cpp (after, excerpt)
Cli::Cli(RegistrationService& service, const EventRepository& events,
         std::ostream& out)
    : service_{service}, events_{events}, out_{out},
      commands_{
          {"list",     [this](const Args& a) { list(a); }},
          {"register", [this](const Args& a) { registerStudent(a); }},
          {"cancel",   [this](const Args& a) { cancel(a); }},
          {"checkin",  [this](const Args& a) { checkIn(a); }},
          {"help",     [this](const Args& a) { help(a); }},
      } {}

void Cli::handleCommand(const std::string& line) {
    const Args args = splitWords(line);
    const std::string cmd = args.empty() ? std::string{} : args[0];
    const auto it = commands_.find(cmd);
    if (it == commands_.end()) {
        out_ << "unknown command: " << cmd << '\n';
        return;
    }
    it->second(args);
}

void Cli::cancel(const Args& args) {
    const auto se = studentAndEvent(args, "cancel <student> <event>");
    if (!se) return;
    const Event event = events_.findById(se->eventId);
    if (!isCancellationOpen(event, Clock::now())) {
        out_ << "too late to cancel\n";
        return;
    }
    service_.cancel(se->studentId, se->eventId);
    out_ << "cancelled\n";
}
```

Reverse engineering: recovering the design from the code



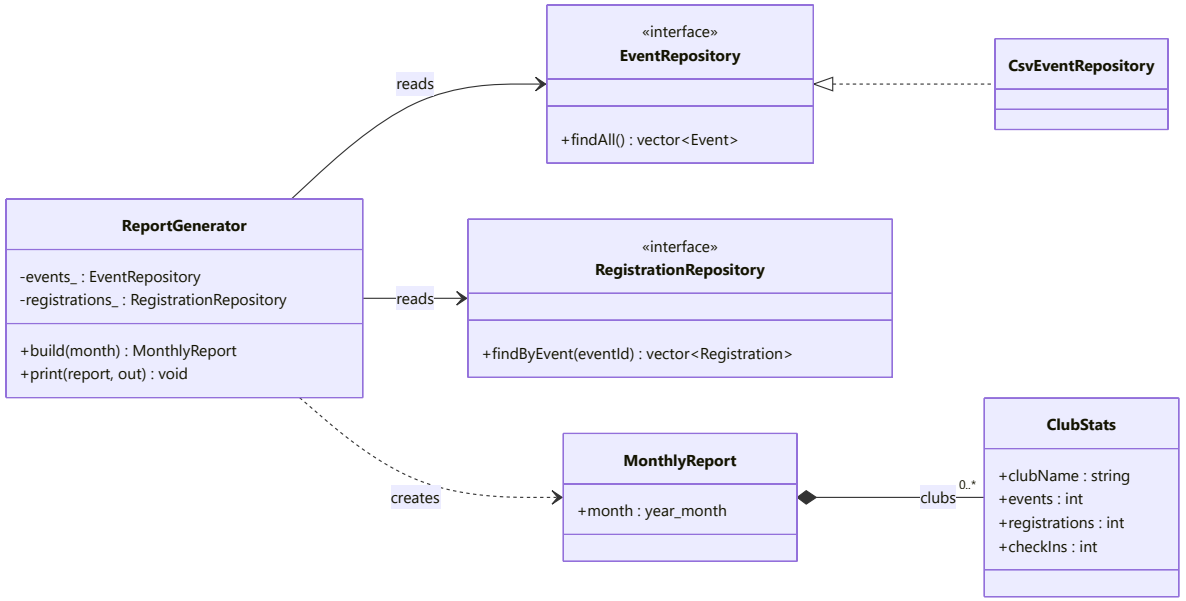
Terms after Chikofsky and Cross (1990). Reverse engineering *analyses* a system to build higher-level descriptions; it does not change the system.

- **Redocumentation**: produce missing documentation at the same level (API docs from headers). **Design recovery**: rebuild class and sequence diagrams and the reasons behind them.
- Sources: the code, the build files, the tests, the Git history, issue tracker, and the people who remember.
- **Characterization tests** (Feathers, 2004) record what the code does today, right or wrong, so you can change it safely.

```
# who changed this function, when, and why?
git log -L :handleCommand:src/Cli.cpp --oneline
git blame -L 40,80 src/Cli.cpp

# API documentation and class graphs from the headers
doxygen -g Doxyfile      # then set EXTRACT_ALL = YES
doxygen Doxyfile         # HAVE_DOT = YES draws class graphs

# where are the complex parts? (CC above 10)
lizard src/ -C 10 -w
```

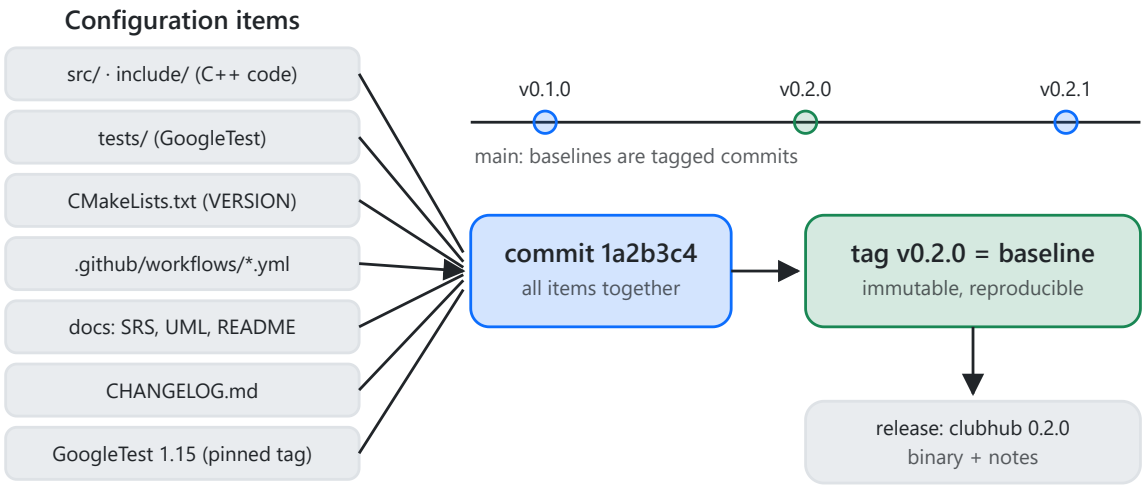
Worked example: recover a class diagram from undocumented C++

```cpp
// include/clubhub/ReportGenerator.hpp
struct ClubStats {
    std::string clubName;
    int events = 0;
    int registrations = 0;
    int checkIns = 0;
};

struct MonthlyReport {
    std::chrono::year_month month;
    std::vector<ClubStats> clubs;
};

class ReportGenerator {
public:
    ReportGenerator(const EventRepository& events,
                    const RegistrationRepository& regs);
    MonthlyReport build(std::chrono::year_month m) const;
    void print(const MonthlyReport& r,
               std::ostream& out) const;
private:
    const EventRepository& events_;
    const RegistrationRepository& registrations_;
};

// include/clubhub/CsvEventRepository.hpp
class CsvEventRepository : public EventRepository {
    // ...
};
```



C++ construct	UML element (UML 2.5.1)
class with only pure virtual functions	«interface»
`public` inheritance from an interface	realization (dashed line, hollow triangle)
member held by value, or `std::vector` of values	composition (filled diamond), with multiplicity
reference or pointer member	association (navigable arrow)
only used as parameter or return type	dependency (dashed arrow)

Configuration management: know exactly what you shipped



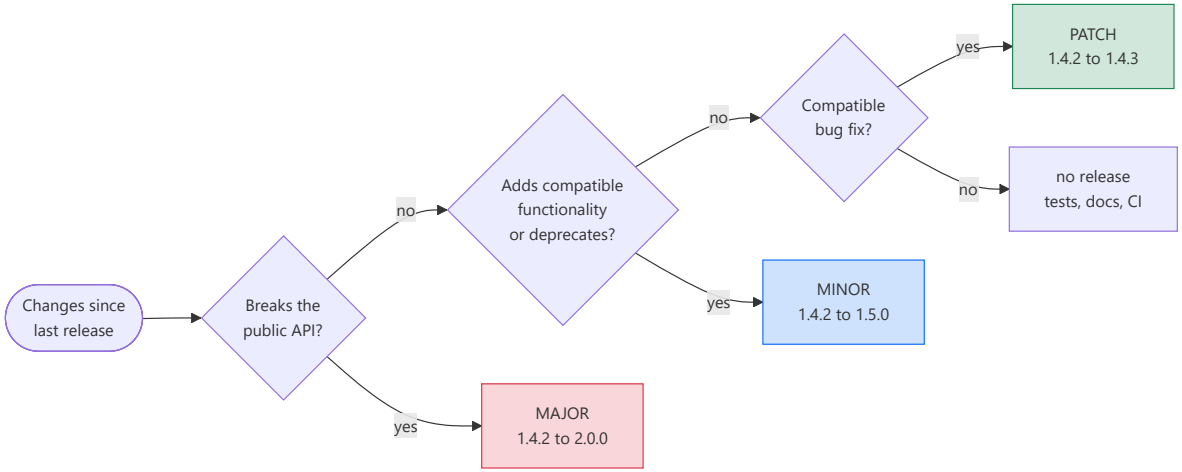
SWEBOK Guide v4.0, Software Configuration Management · IEEE 828-2012

- A **configuration item** (CI) is anything whose version matters to rebuild a release: code, tests, build files, CI workflows, documents, pinned dependencies.
- A **baseline** is an approved, frozen set of CI versions. In Git: a tagged commit.
- Four activities (SWEBOK Guide v4.0): **identification** (what is a CI, how is it named), **change control** (who may change it: pull requests and reviews), **status accounting** (what is in which version), **audit** (does the release match its baseline?).

```
git tag -a v0.2.0 -m "Sprint 2 release"
git push origin v0.2.0
git describe --tags   # v0.2.0-3-g1a2b3c4
# = 3 commits after v0.2.0, at commit 1a2b3c4
```

⚠ `GIT_TAG main` in `FetchContent` is not a baseline: the build changes when someone else pushes. Pin a release tag.

Semantic versioning: the number tells users what changed



- SemVer 2.0.0: version `MAJOR.MINOR.PATCH`. The rules only work once you have declared a **public API**.
- Club Hub's public API: the CLI commands and their output, the CSV file formats, and the headers in `include/clubhub/`.
- When several changes go into one release, the **largest** bump wins.
- `0.y.z` is initial development: anything may change. Club Hub stays at 0.x until the week-14 submission, which can become `1.0.0`.
- Pre-release and build labels: `1.0.0-rc.1` comes before `1.0.0`; `1.0.0+build.57` adds metadata that does not affect order.

> Once a version is released, its contents must never change. A mistake in `1.4.3` is fixed in `1.4.4`, not by moving the tag.

Worked example: four changes, four version decisions

Suppose Club Hub is at **1.3.2**, a year after the course. Each change is judged on its own:

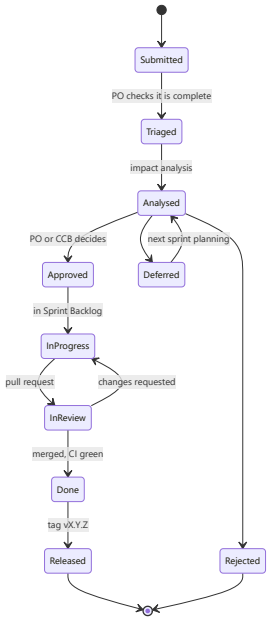
Change	Public API effect	Next
Fix: cancelling exactly 24 h before the start was rejected	none; behaviour now matches the documented rule	PATCH 1.3.3
Add a waiting list, switched on per event	new command `waitlist`; old commands and files unchanged	MINOR 1.4.0
Rename CSV column `student_id` to `studentId`, old files no longer load	breaks every existing data file	MAJOR 2.0.0
Refactor `handleCommand` into a command map	none; output identical	PATCH 1.3.3, or no release on its own

⚠ The CSV rename becomes MINOR if the new version still reads the old column name. Compatibility is a design choice, and it is usually cheaper than a MAJOR release.

```
# CMakeLists.txt: the single source of the version number
cmake_minimum_required(VERSION 3.28)
project(clubhub VERSION 1.4.0 LANGUAGES CXX)

configure_file(include/clubhub/Version.hpp.in
               ${PROJECT_BINARY_DIR}/generated/clubhub/Version.hpp
               @ONLY)
target_include_directories(clubhub_core PUBLIC
                           ${PROJECT_BINARY_DIR}/generated)
```

```
// include/clubhub/Version.hpp.in
#pragma once
#include <string_view>

namespace clubhub {
inline constexpr std::string_view kVersion = "@PROJECT_VERSION@";
}

// src/main.cpp: clubhub --version  prints  clubhub 1.4.0
std::cout << "clubhub " << clubhub::kVersion << '\n';
```

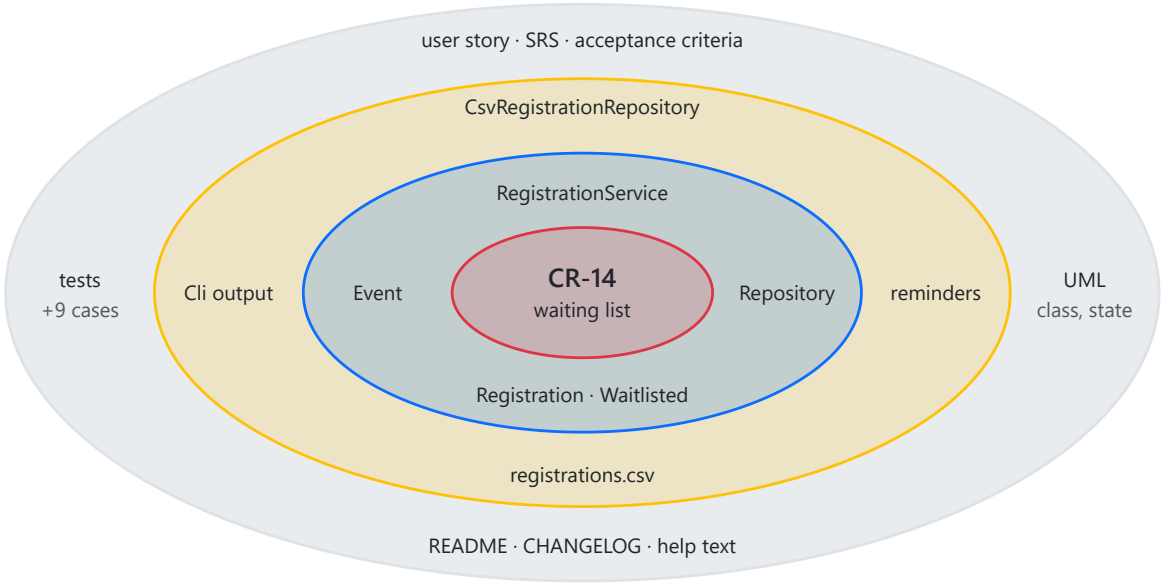
A change request has a life cycle, and every state has an owner




```
CHANGE REQUEST CR-14                              Status: Approved
Title       Waiting list for full events
Requested   Robotics Club leader, week 12, after the Sprint 1 review
Category    Perfective (additive) · MoSCoW: Should
Problem     Popular events fill within minutes. When a student cancels,
            the seat stays empty because nobody knows it is free.
Proposal    When an event is full, "register" puts the student on a
            waiting list (first come, first served). A cancellation
            promotes the first waitlisted student to Registered.
Acceptance  Given a full event, when a student registers,
            then the status is Waitlisted and the position is shown.
Impact      See impact analysis IA-14: about 14 h, 5 classes, 9 tests
Size        M (5 story points) · Urgency: medium
Decision    Approved by the Product Owner for Sprint 2; target v0.2.0
```

- **Who may approve?** In Scrum the Product Owner orders the backlog; changes that affect other systems or the release date may need a **change control board** (CCB): here PO, Scrum Master and the instructor.
- Rejected and deferred requests are recorded with a reason, never silently dropped. The IT Project Management course covers integrated change control in depth.

Change impact analysis: follow the ripples before you change



- **Change impact analysis**: identify the potential consequences of a change, or estimate what must be modified, *before* making it (Bohner and Arnold, 1996).
- **Ring 1** (red to blue): code that changes directly, the *starting impact set*. **Ring 2** (yellow): code and data that depend on it. **Ring 3** (grey): tests, models and documents.
- Two techniques: **traceability** (requirement → design → code → test links from Lab-05 and Lab-06) and **dependency analysis** (who includes, calls or stores what).
- After the change, compare the estimated set with the files the pull request really touched; the difference improves the next estimate.

```
# dependency analysis with plain tools
grep -rln "RegistrationStatus" include src tests
grep -rn "registerStudent(" src tests | wc -l
```

Worked example: impact analysis IA-14 "allow events to have a waiting list"

Artifact	Affected items	Change	Est.
Requirements	capacity rule; new user story "join the waiting list" with Given/When/Then; SRS section on registration	amend, add	1 h
UML	class diagram (`Event::waitlistEnabled`, repository operation); state diagram of `Registration`: Waitlisted → Registered on promotion, Waitlisted → Cancelled	update	1 h
Code	`include/clubhub/Event.hpp`, `RegistrationService.hpp/.cpp` (`registerStudent` returns the status, `cancel` promotes), `RegistrationRepository.hpp` (`findWaitlisted(eventId)` in order), `CsvRegistrationRepository.cpp` (read and write `waitlisted`), `src/Cli.cpp` (messages, `waitlist` command)	modify	5 h
Tests	`RegistrationServiceTest` +6 (full → waitlisted, promotion on cancel, FIFO order, no duplicate on the list, cancel while waitlisted, list disabled); `CsvRegistrationRepositoryTest` +2 round trips; `CliTest` +1	add	3 h
Docs	`README.md` usage, `help` text, `CHANGELOG.md` "Added"	update	0.5 h
Release	version 0.1.0 → 0.2.0 (MINOR: new, backward-compatible feature)	bump, tag	0.5 h
Subtotal · plus 30 % for uncertainty (first time the team touches the promotion logic)			**11 h → 14 h**

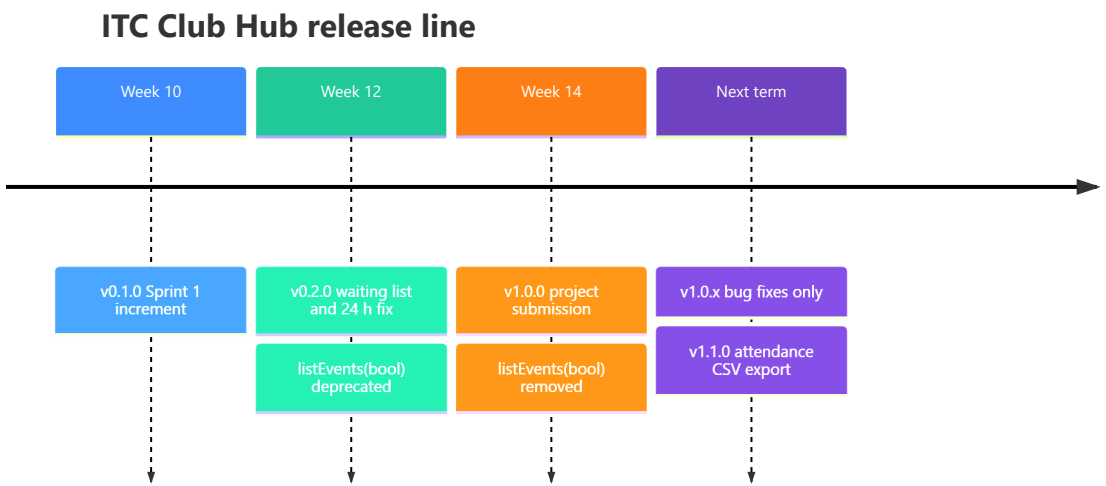
Risks found by the analysis

1. Data files written by v0.1.0 must still load: `waitlisted` is a new value in an existing column, so old files stay valid.
2. Promotion and cancellation must not both fill the last seat: one function, one test for the race-free order.
3. The reminder job must skip waitlisted students.

The enum value `RegistrationStatus::Waitlisted` already exists in the Lab-06 model, so the class diagram change is small. Good models make impact analysis cheaper.

Output of the analysis: the affected list, an estimate with its uncertainty, the risks, the version decision, and a recommendation (approve for Sprint 2) that goes back into CR-14.

Release management: planned releases and a polite deprecation



- **Release management** decides what goes into a release, builds and tests it from a baseline, versions and tags it, writes the notes, distributes it, and supports it afterwards.
- **Cadence**: time-based (every sprint, every term) or feature-based (when CR-14 is done). Student teams do best with time-based releases at the end of each sprint.
- **Deprecation** is how you remove something without surprising users: *announce* it (changelog, release notes), *warn* (compiler warning, runtime message), keep it working for at least one MINOR release, and *remove* it only in the next MAJOR release.
- **Support window**: say which versions still get fixes. For Club Hub after the course: only the latest 1.0.x.

```
// v0.2.0: the old overload still works, but warns
[[deprecated("use listEvents(EventFilter); removed in 1.0.0")]]
std::vector<Event> listEvents(bool onlyUpcoming) const;

std::vector<Event> listEvents(const EventFilter& filter) const;
```

A caller now sees a compiler warning such as *'listEvents' is deprecated: use listEvents(EventFilter)*, with the version that removes it.

Worked example: `CHANGELOG.md` and a tag-triggered release

```
# Changelog
All notable changes to ITC Club Hub are documented here.
The format is based on Keep a Changelog 1.1.0, and this
project adheres to Semantic Versioning.

## [Unreleased]

## [0.2.0] - 2026-11-20
### Added
- Waiting list for full events; a cancellation promotes
  the first waitlisted student (CR-14).
- `waitlist <event>` command shows the queue.
### Changed
- `handleCommand` uses a command map (no behaviour change).
### Deprecated
- `listEvents(bool)`: use `listEvents(EventFilter)`.
  Will be removed in 1.0.0.
### Fixed
- Cancelling exactly 24 h before the start was rejected
  (CR-13).

## [0.1.0] - 2026-11-06
### Added
- Register, cancel and check in from the command line.
```

Six section types: Added, Changed, Deprecated, Removed, Fixed, Security. Write the entry in the same pull request as the change, under *Unreleased*.

```yaml
# .github/workflows/release.yml
name: release
on:
  push:
    tags: ['v*.*.*']
permissions:
  contents: write
jobs:
  release:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Tag must match the CMake project version
        run: grep -q "VERSION ${GITHUB_REF_NAME#v}" CMakeLists.txt
      - run: cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
      - run: cmake --build build
      - run: ctest --test-dir build --output-on-failure
      - name: Release notes = this version's CHANGELOG section
        run: |
          awk -v v="${GITHUB_REF_NAME#v}" \
            '$0 ~ "^## \\[" v "\\]" {p=1; next} /^## \[/ {p=0} p' \
            CHANGELOG.md > notes.md
      - run: gh release create "$GITHUB_REF_NAME" --notes-file notes.md
        env:
          GH_TOKEN: ${{ github.token }}
```

The release job refuses a tag that does not match `project(... VERSION ...)` and never publishes a red build. Release notes come from the changelog, so there is one source of truth.

Worked example: cyclomatic complexity of `handleCommand`, counted by hand

🧮 **Cyclomatic complexity** (McCabe, 1976) = number of independent paths through a function. By hand: start at **1** and add 1 for each `if`, `else if`, `for`, `while`, `case`, `catch`, `&&`, `||` and `?:`. It is also the minimum number of test cases for full branch coverage of the function.

Function	Decision points	CC
`handleCommand` (before)	5 `if`/`else if` on `cmd` + 1 `for` + 3 usage `if` + 1 deadline `if` = 10	**11**
`handleCommand` (after)	1 `?:` + 1 `if` = 2	**3**
`cancel`, `studentAndEvent`	2 `if` each	3
`list`, `registerStudent`, `checkIn`	1 each	2
`help`, `isCancellationOpen`	none	1

```
$ lizard src/Cli.cpp                # before
  NLOC    CCN   token  PARAM  length  location
    41     11    283      1      41 Cli::handleCommand@16-56@src/Cli.cpp
$ lizard src/Cli.cpp                    # after (largest function)
    10      3     88      1      10 Cli::handleCommand@71-80@src/Cli.cpp
```

- The **maximum** per function dropped from 11 to 3; the **total** grew slightly (16 over seven functions). Refactoring does not remove the decisions, it spreads them into small, named, separately testable pieces.
- A common threshold is CC ≤ 10 per function (McCabe's original advice); `lizard` warns above 15 by default.

Maintainability (ISO/IEC 25010:2023)	Example metric	Club Hub target
Modularity	files touched per change request	typically 3 or fewer
Reusability	code used by both CLI and tests	domain has no I/O
Analysability	CC and length per function	CC ≤ 10, ≤ 40 lines
Modifiability	lead time from CR approval to release	within one sprint
Testability	line coverage with gcovr (Lab-09)	≥ 80 % of `src/`

⚠ A metric is a question, not a verdict. Watch trends per release, and never reward people for a number: CC is easy to lower by making code worse.

A maintenance plan: who keeps Club Hub alive after week 14?

```
# ITC Club Hub · Maintenance plan (v1.0, week 12)

## Scope and support
- Supported release: the latest 1.x; fixes ship as 1.x.PATCH.
- Compilers: GCC 13+, Clang 17+, MSVC 2022+ (C++20).
- Platforms: Windows 11, Ubuntu 24.04, macOS 14.

## Roles (after the course)
- Maintainer: the clubs office student assistant, named per term.
- Bugs: GitHub issue with the bug template; triage every Monday.
- Response: blocker 2 days · major 1 week · minor next release.

## Routine work
- Monthly: update the GoogleTest tag, run CI, read release notes.
- Each term: build with the newest lab compilers (adaptive work).
- 20 % of maintenance time goes to the technical-debt register.

## Effort and end of life
- ACT 20 % x 300 development hours = 60 h per year (5 h/month).
- Review every year; retire when the institute adopts a new system.
```

- ISO/IEC/IEEE 14764:2022 asks for a **maintenance plan** before delivery: scope of support, who maintains, resources, the process for requests, and how the handover works.
- **Estimating effort** with annual change traffic (Boehm, COCOMO, 1981): annual maintenance effort = ACT × development effort, where ACT is the fraction of the code expected to change each year.

Club Hub estimate

Development effort: 5 students × 60 h = 300 h

Expected ACT: about 20 % (new clubs' requests, compiler updates)

Maintenance: 0.20 × 300 h = **60 h per year**, about 5 h per month

Plan the handover. Without a named owner, a README that explains how to build and test, and green CI, a student project dies the week after the demo.

Keeping a system healthy: do this, avoid that

1 Pin behaviour first

Before changing code you do not fully understand, write tests (or characterization tests) that record what it does today.

2 Small commits, two hats

One refactoring per commit, tests green each time; behaviour changes and bug fixes go in their own commits.

3 Every change is a request

Log it, classify it, analyse its impact and estimate it before approving it. Keep rejected requests with their reason.

4 One version, one changelog

The version lives in `CMakeLists.txt`, tags follow SemVer, and each pull request adds its line under *Unreleased*.

5 Pay debt every sprint

Keep a register with principal and interest, and reserve a fixed share of each sprint for the highest-interest items.

6 Plan for the next maintainer

A named owner, a README that builds from scratch, supported compilers written down, CI that stays green.

✕ The big-bang rewrite

Throwing away working legacy code and rebuilding it all at once. Prefer wrap, refactor and strangler-fig replacement.

✕ "Refactoring" without tests

Without a safety net it is just editing and hoping. A red test during a refactoring means undo, not "fix the test".

✕ Silent breaking changes

Removing a command or changing a file format in a MINOR or PATCH release, without deprecation or a changelog entry.

Chapter quiz 10 questions

1. The institute changes student IDs from 9 to 10 characters, and Club Hub now rejects every new ID. Which maintenance category is the change to Club Hub?

- Corrective
- Adaptive
- Perfective
- Preventive

2. A code review finds that the seat counter can go negative if `cancel` is called twice, although no user has ever hit it. Fixing it now is...

- Corrective maintenance, because every fault fix is corrective
- Preventive maintenance: a latent fault fixed before it becomes a failure
- Adaptive maintenance
- Perfective maintenance

3. What does Lehman's law of increasing complexity state?

- As a system changes, its complexity grows unless work is done to reduce it
- A system becomes simpler as its bugs are fixed
- The size of a system stays constant over its releases
- Every system must be rewritten after five years

4. A team says: "We store data in CSV for the demo, and we logged a backlog item to move to JSON next sprint." Which kind of technical debt is this?

- Reckless and deliberate
- Reckless and inadvertent
- Prudent and deliberate
- Prudent and inadvertent

5. Which statement describes refactoring?

- Adding a feature and cleaning the code in the same commit
- Changing the internal structure without changing the observable behaviour, checked by tests
- Rewriting the system in a new language
- Fixing a bug that a user reported

6. Adding one new registration status forces edits in six different files. Which code smell is this?

- Long Parameter List
- Shotgun Surgery
- Feature Envy
- Data Class

7. Club Hub is at version 1.4.2. You add a CSV export command, and nothing existing changes. What is the next version under SemVer?

- 1.4.3
- 1.5.0
- 2.0.0
- 1.5.2

8. A C++ function contains one `for` loop and two `if` statements, and one of the `if` conditions uses `&&`. What is its cyclomatic complexity?

- 3
- 4
- 5
- 6

9. What should a change impact analysis deliver before a change request is approved?

- The finished code for the change
- The affected requirements, models, code, tests and documents, with an effort estimate and the risks
- Only the next version number
- A retrospective of the last sprint

10. In a Keep a Changelog file, where do you announce that `listEvents(bool)` will be removed in the next major version?

- Under Removed
- Under Deprecated
- Under Fixed
- Nowhere: removing it later is enough

WRAP-UP

Summary

Maintenance is every change after delivery, and it is most of the life-cycle cost.	Legacy options run from retain and wrap to refactor, replatform and replace; prefer incremental change.	Reverse engineering recovers models from code; characterization tests pin what it does today.
ISO/IEC/IEEE 14764: corrective and adaptive are reactive; preventive and perfective are proactive.	Technical debt has principal and interest: register it and pay the highest interest first.	Baselines, SemVer and a changelog make every release traceable and every change honest.
Lehman: a used system must keep changing, and its complexity grows unless you work against it.	Refactor in small steps behind green tests, never mixed with behaviour changes.	Change requests go through impact analysis; plan maintenance with metrics and a named owner.

Open Lab-10: 5 tasks + 1 challenge

Next chapter: 11 · Emerging Trends in Software Engineering