INTRODUCTION TO SOFTWARE ENGINEERING · CHAPTER 09

Software Testing and Quality Assurance

Testing finds defects; quality assurance prevents them. This chapter introduces test design, automated testing and the wider practices that build quality into ITC Club Hub.

ISO/IEC 25010:2023 | GoogleTest 1.15 · gcov · gcovr | clang-tidy · cppcheck · C++ Core Guidelines | C++20 · CMake · CTest

AGENDA

What we will cover

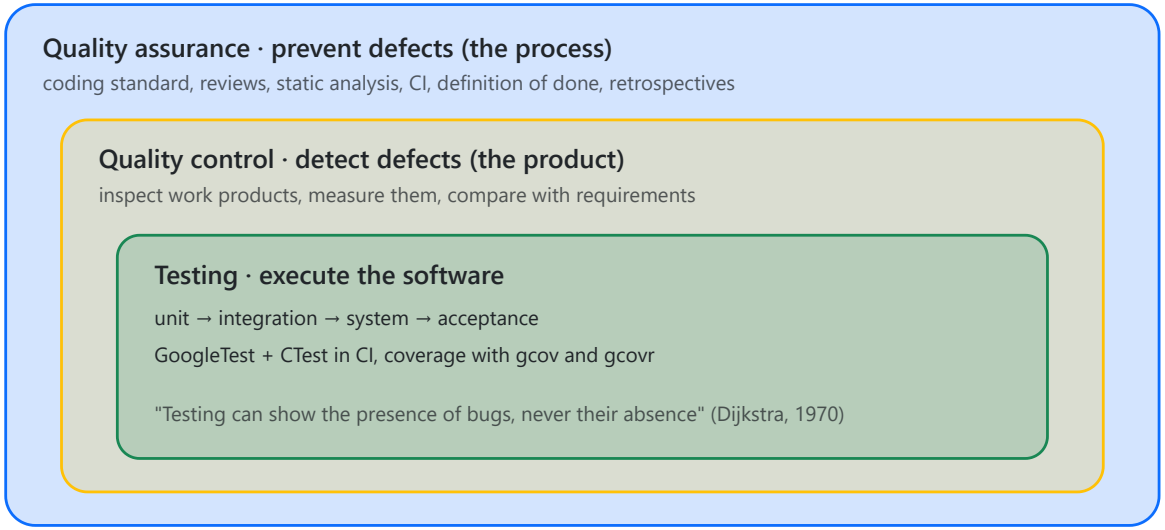
1. Verification and validation; errors, faults and failures
2. Testing levels: unit, integration, system, acceptance
3. Testing types: functional, non-functional, regression
4. Black-box test design: equivalence partitioning and boundary value analysis
5. White-box testing: statement and branch coverage
6. Unit testing with GoogleTest and test-driven development
7. Test plans, test cases and test reports
8. Defect life cycle and writing good bug reports
9. Quality assurance: reviews, inspections, static analysis (clang-tidy, cppcheck), coding standards (C++ Core Guidelines)
10. Software quality models (ISO/IEC 25010) and quality metrics
11. Chapter Quiz
12. Lab-09

Click any item to jump directly to that topic.

Why a whole chapter on quality? Sprint 2 is where bugs get expensive

- In Chapter 08 your pipeline started to compile ITC Club Hub and run CTest on every push. A green pipeline only means something if the **tests are well chosen**.
- **Testing** executes the software to find defects. **Quality control** checks any work product (code, SRS, diagrams) against its requirements. **Quality assurance** improves the *process* so that fewer defects are made at all.
- The business rules of Club Hub (capacity, duplicate registration, the 24-hour cancellation deadline, events in the past) are exactly where small mistakes hide.
- Exhaustive testing is impossible: we need **test design techniques** to choose a small set of tests that finds many defects.

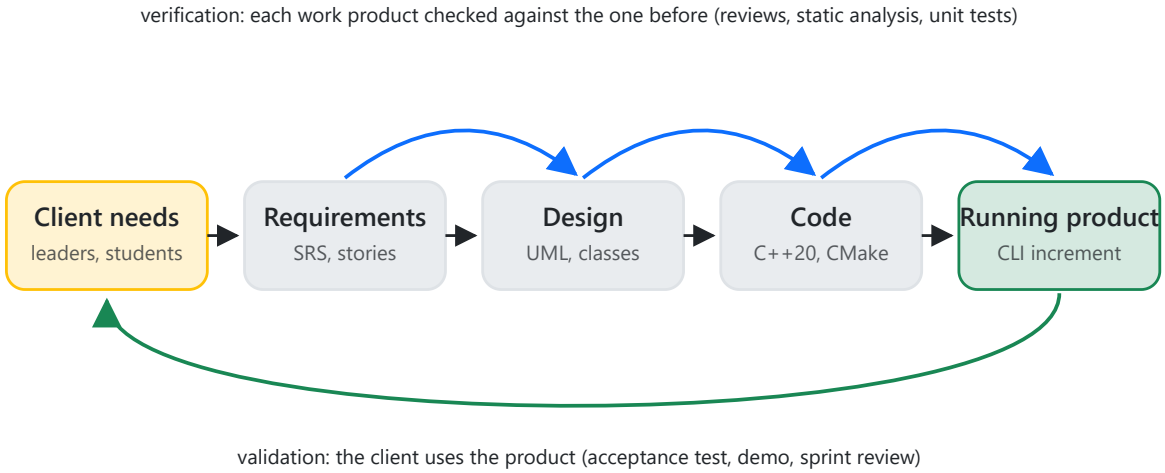
📖 SWEBOK Guide v4.0 treats *Software Testing* and *Software Quality* as two separate knowledge areas; this chapter surveys both. The Software Engineering course covers automated testing in depth.



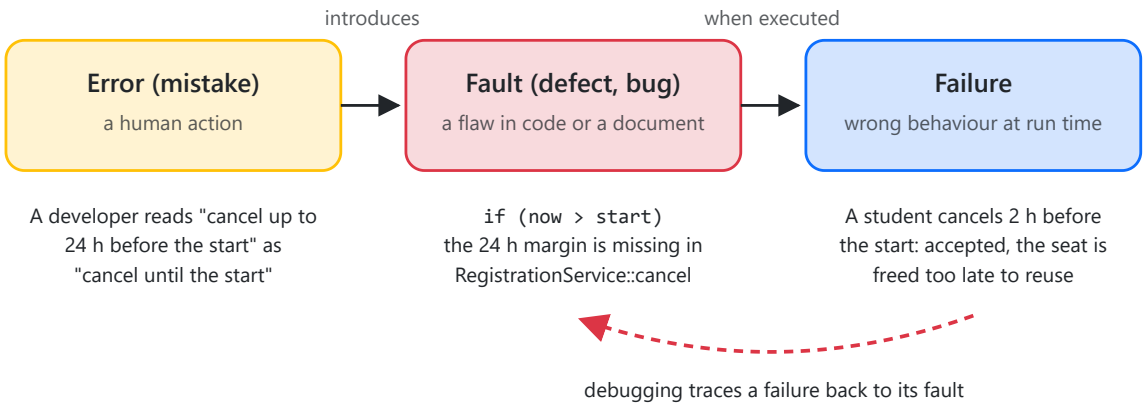
Building the product right, and building the right product

- **Verification**: does each work product match the one it was derived from? Design against SRS, code against design, behaviour against the stated rule. "Are we building the product right?"
- **Validation**: does the product meet the client's real needs in its real use? "Are we building the right product?" (Boehm, 1979)
- *Verification example*: a unit test proves that `cancel` rejects a request 23 hours before the start, as FR-12 says.
- *Validation example*: at the sprint review a club leader says students on the waiting list must also be able to cancel. Every test passed, yet a need was missing.

Both are needed: perfect verification of a wrong requirement delivers the wrong product, correctly. Sources: SWEBOK Guide v4.0; IEEE 1012 (system, software and hardware V&V).



A human error leaves a fault in the code; running it may cause a failure

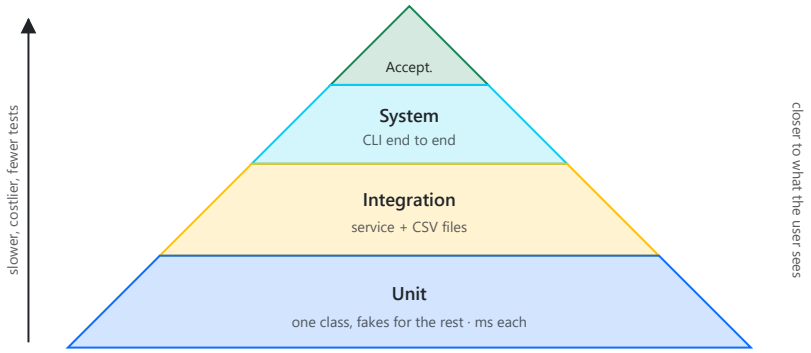


root-cause analysis asks why the error happened: unclear SRS wording? no review? no boundary test?

```
// FR-12: cancel allowed up to 24 h before start
void RegistrationService::cancel(
        const std::string& studentId, int eventId) {
    const Event& event = events_.get(eventId);
    // FAULT: should be event.start() - 24h
    if (clock_.now() > event.start()) {
        throw RuleViolation{Rule::CancellationClosed};
    }
    regs_.setStatus(studentId, eventId,
                    RegistrationStatus::Cancelled);
}
```

⚠ A fault that is never executed with a triggering input causes no failure: a test at 72 h before the start passes with this bug. Only a test *inside* the last 24 hours reveals it. Terms from ISO/IEC/IEEE 24765 and SWEBOK Guide v4.0.

Four levels, from one function to the client's acceptance

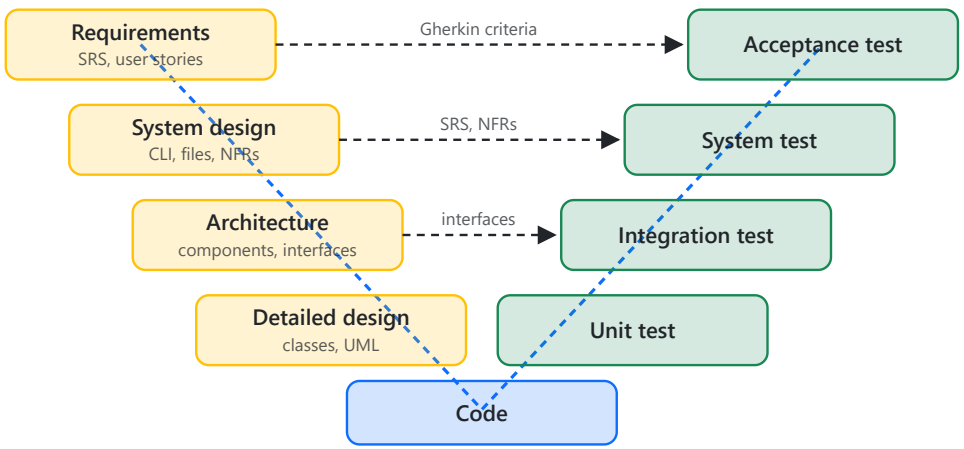


Test pyramid after Cohn (2009): most tests at the bottom, where they are fast and pinpoint the fault.

Level	What is tested	ITC Club Hub example	Who
Unit	One function or class in isolation	`RegistrationService::cancel` with in-memory repositories and a fake clock	Developer
Integration	The interfaces between units or components	`RegistrationService` + `CsvRegistrationRepository` writing a real temporary file	Developer
System	The complete system against the SRS, including non-functional requirements	Script runs `clubhub register`, `cancel`, `check-in` and compares the output files	Team, test lead
Acceptance	Whether the client accepts it for use	Gherkin scenarios of Lab-05 run with the Product Owner and the client at the sprint review	Client, PO

💡 An inverted pyramid (mostly manual system tests, few unit tests) is the "ice-cream cone" anti-pattern: slow feedback and failures that do not say where the fault is. Source: SWEBOK Guide v4.0, Software Testing KA, test levels.

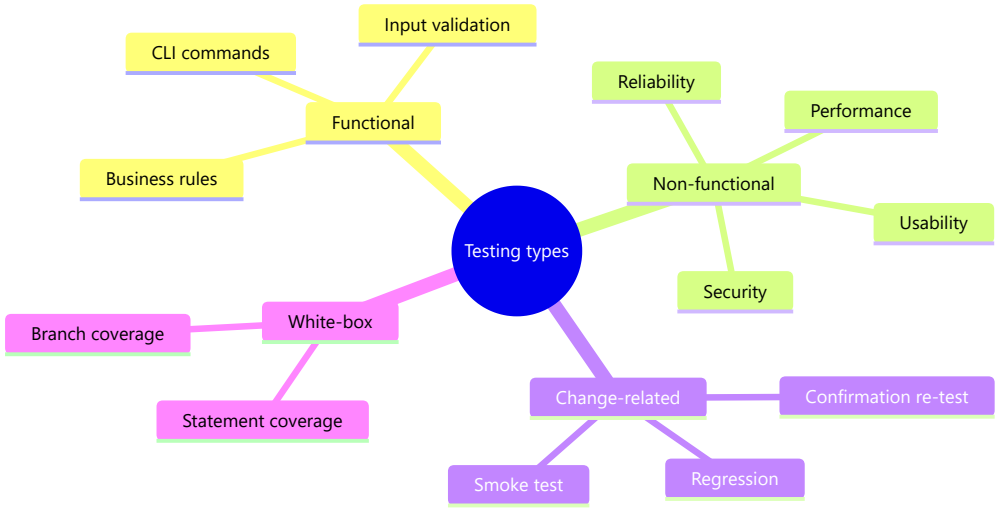
Each level is planned from a document on the left of the V



The V-model from Chapter 04 pairs each specification with the test level that checks it (dashed: the *test basis*). In Scrum the same pairs exist, but inside every sprint instead of once per project.

```cpp
// tests/csv_registration_repository_it.cpp
// Integration: real file system, no fakes
TEST(CsvRegistrationRepositoryIT, SavesAndReloads) {
    namespace fs = std::filesystem;
    const fs::path dir =
        fs::temp_directory_path() / "clubhub-it";
    fs::create_directories(dir);
    const fs::path file = dir / "registrations.csv";
    {
        clubhub::CsvRegistrationRepository repo{file};
        repo.save({"e20230001", 1,
                   clubhub::RegistrationStatus::Registered});
    }
    // repo destroyed above: the reload must read the file
    clubhub::CsvRegistrationRepository reloaded{file};
    const auto r = reloaded.find("e20230001", 1);
    ASSERT_TRUE(r.has_value());
    EXPECT_EQ(r->status, clubhub::RegistrationStatus::Registered);
    fs::remove_all(dir);
}
```

Levels say *where* we test; types say *what quality* we test for



Type	Question it answers	ITC Club Hub example
Functional	Does it do what the requirement says?	With capacity 30, the 31st registration is rejected
Performance	How fast, how much?	`clubhub list-events` loads 5000 events from CSV in under 1 s
Usability	Can a first-year student use it without help?	Three students register for an event; count errors and questions
Security	Can data leak or be corrupted?	Attendance export contains no phone numbers; a name with a comma does not break the CSV
Regression	Did the change break something that worked?	The whole GoogleTest suite runs in CI on every pull request
Confirmation	Is this bug really fixed?	Re-run the test that exposed bug #47 after the fix
Smoke	Is the build worth testing further?	`clubhub --version` and `list-events` after each build

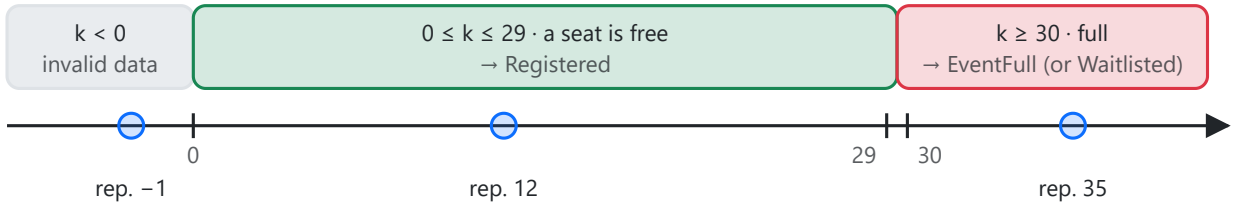
💡 Types and levels are independent: a performance test can be a unit benchmark or a system test. Regression testing is cheap only when it is automated. Classification after ISTQB CTFL v4.0 (functional, non-functional, white-box, change-related).

Equivalence partitioning: one test per class of inputs that behave the same

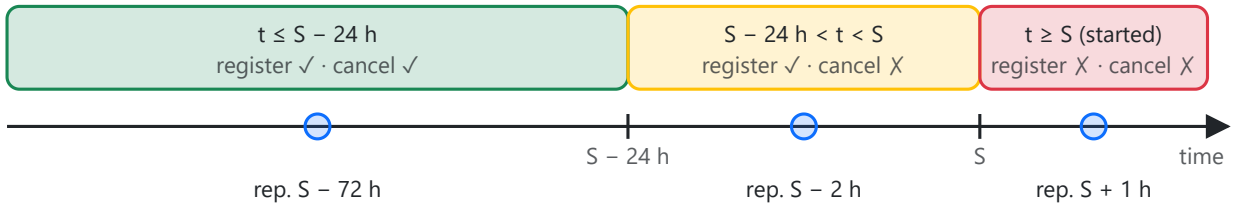
- **Black-box** techniques derive tests from the specification only, without reading the code.
- Split each input or condition into **partitions** (equivalence classes): values the rule treats the same way. Include *invalid* partitions.
- Pick **one representative** per partition; if one value in a class fails, the others probably do too.
- Non-numeric conditions partition too: student already registered (yes / no), event exists (yes / no), club approved (yes / no).

Rules from the case study: capacity cannot be exceeded; cancel up to 24 h before the start; events that have started cannot be registered for.

A · Seats already taken k when the request arrives (capacity C = 30)



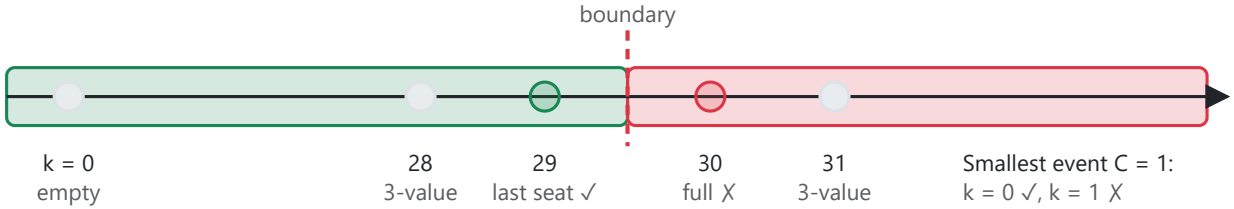
B · Time t of the request relative to the event start S



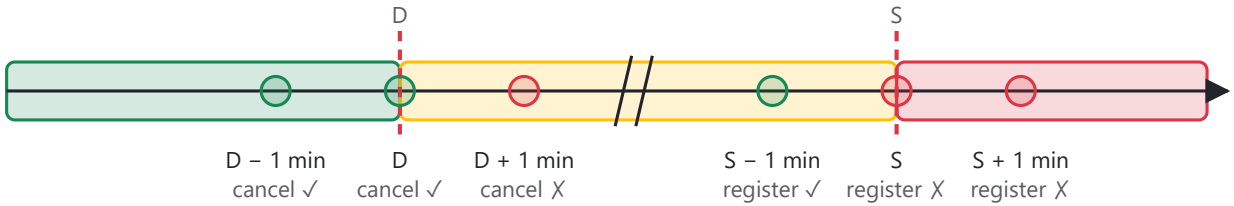
one representative per class is enough if the class really behaves the same (the equivalence hypothesis)

Boundary value analysis: faults cluster at the edges of partitions

A · Capacity C = 30: the boundary lies between k = 29 and k = 30



B · Time: cancel closes at D = S − 24 h (inclusive), register closes at S



axis B not to scale · the faults hide here: > instead of >=, 24 h instead of 24 h − 1 min, off-by-one counts

- A **boundary value** is the smallest or largest value of an ordered partition.
- **2-value BVA**: test the boundary and its closest neighbour in the next partition (29 and 30). **3-value BVA** also tests the value on the other side (28, 29, 30 and 29, 30, 31).
- For time, choose the *resolution* first: the CLI accepts minutes, so the neighbour of D is D ± 1 min.
- Ambiguous words become visible: is "up to 24 h" inclusive? Ask the Product Owner, write the answer into the acceptance criteria, then test it.

Techniques as described in ISTQB CTFL v4.0 and SWEBOK Guide v4.0 (input domain-based techniques).

Test design for `registerStudent` and `cancel`: 11 cases, each with a reason

TC	Technique · class or boundary	Input (event state, student, time t)	Expected result	FR
R1	EP · seat free (n seats left)	C = 30, k = 12, not registered, t = S − 72 h	Registered; 13 seats taken	FR-11
R2	BVA · 1 seat left	C = 30, k = 29	Registered; event now full	FR-11
R3	BVA · 0 seats left	C = 30, k = 30	`RuleViolation` EventFull (Waitlisted once the Should story is built)	FR-11
R4	BVA · smallest event	C = 1, k = 0, then a second student	First Registered; second EventFull	FR-11
R5	EP · invalid capacity	create event with C = 0	`Event` constructor throws `std::invalid_argument`	FR-05
R6	EP · already registered	same student, same event, twice	`RuleViolation` AlreadyRegistered; still 1 row	FR-11
R7	EP · event in the past	t = S + 1 h	`RuleViolation` EventInPast	FR-11
R8	BVA · registration closes at S	t = S (and S − 1 min: Registered)	`RuleViolation` EventInPast	FR-11
R9	BVA · 24 h rule does not apply	register at t = S − 24 h exactly	Registered (the 24 h rule is for cancel only)	FR-11
C1	BVA · cancel on the deadline	registered, cancel at t = S − 24 h	Cancelled; seat free again	FR-12
C2	BVA · cancel just after it	registered, cancel at t = S − 24 h + 1 min	`RuleViolation` CancellationClosed; still Registered	FR-12

✓ "Capacity 0 / 1 / n" means three different things: an invalid event (R5), the last seat (R2) and the normal case (R1). R4 covers the smallest valid capacity.

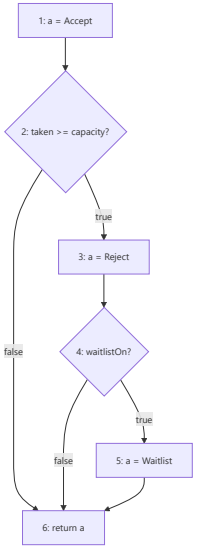
⚠ R9 catches a classic mix-up: a developer who applies the 24-hour rule to registration too. A good test set also checks what a rule does *not* say.

FR ids are examples; use the ids of your own SRS from Lab-05. Every row becomes a GoogleTest test in Lab-09.

White-box: the code's structure tells us what is still untested

```
enum class Admission {
    Accept, Waitlist, Reject };

// 6 statements, 2 decisions
Admission admit(int taken, int capacity,
                bool waitlistOn) {
/*1*/ auto a = Admission::Accept;
/*2*/ if (taken >= capacity) {
/*3*/     a = Admission::Reject;
/*4*/     if (waitlistOn)
/*5*/         a = Admission::Waitlist;
      }
/*6*/ return a;
}
```

- **Statement coverage** = executed statements / all statements.
- **Branch coverage** = decision outcomes taken / all outcomes (each `if` has a true and a false outcome).

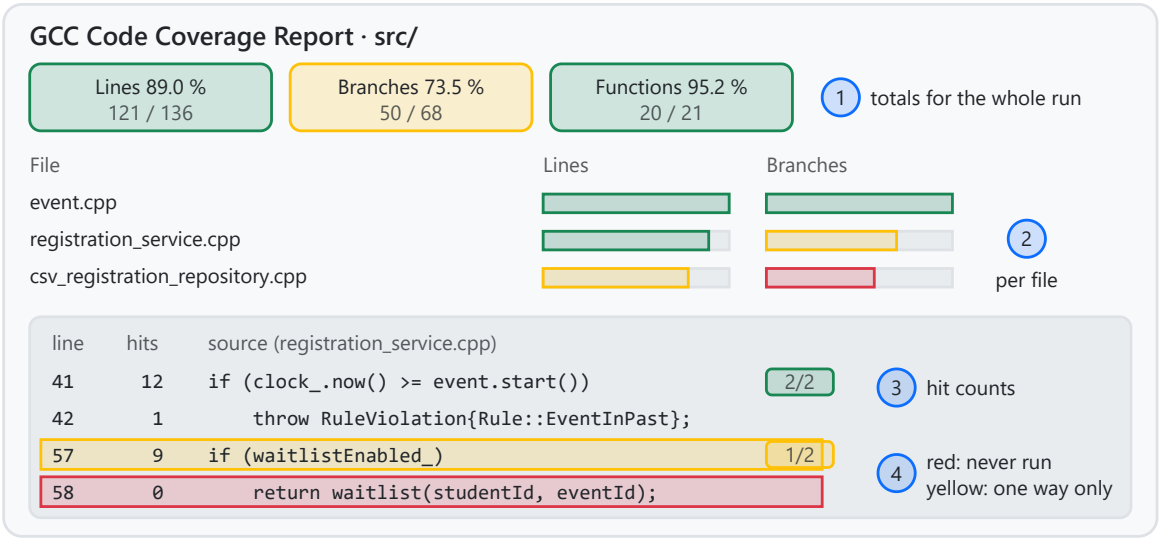


Tests run	Statements	Stmt cov.	Outcomes taken	Branch cov.
T1 (10, 30, false)	1, 2, 6	3/6 = 50 %	D1 false	1/4 = 25 %
T1 + T2 (30, 30, true)	1-6	6/6 = 100 %	D1 false, D1 true, D2 true	3/4 = 75 %
+ T3 (30, 30, false)	1-6	100 %	all four	4/4 = 100 %

⚠ After T2 every line has run, yet the path "full and no waiting list" (D2 false) was never tested. If line 3 were missing, T1 + T2 would still pass. Branch coverage finds this gap; statement coverage does not.

100 % branch coverage implies 100 % statement coverage, not the reverse. Coverage measures the tests, it does not prove the code correct: a test with no assertion still "covers" lines.

Measuring coverage: gcov counts, gcovr reports




```
# 1. build with instrumentation (GCC)
cmake -S . -B build-cov -DCMAKE_BUILD_TYPE=Debug \
      -DCMAKE_CXX_FLAGS="--coverage -O0"
cmake --build build-cov
# 2. run the tests: writes .gcda counters
ctest --test-dir build-cov
# 3. report on our code only
gcovr -r . build-cov --filter src/ \
      --exclude-throw-branches \
      --html-details coverage/index.html \
      --print-summary
```

```
lines: 89.0% (121 out of 136)
functions: 95.2% (20 out of 21)
branches: 73.5% (50 out of 68)
```

Read the red and yellow lines, not the percentage: each one is a test you have not written yet, or code that is unreachable. `--exclude-throw-branches` hides the hidden branches the compiler adds for exceptions. MSVC does not produce gcov data: use the CI job or WSL.

A first GoogleTest file: TEST, EXPECT_* and ASSERT_*

```cpp
// tests/event_test.cpp
#include <gtest/gtest.h>
#include <chrono>
#include <stdexcept>
#include "clubhub/event.hpp"

using namespace std::chrono;
using clubhub::Event;

namespace {
const auto kStart = sys_days{2026y / November / 20} + 18h;
}

// TEST(SuiteName, TestName): one behaviour per test
TEST(EventTest, KeepsTitleAndCapacity) {
    const Event e{1, "C++ Workshop", kStart, "B-201", 30};
    EXPECT_EQ(e.title(), "C++ Workshop");
    EXPECT_EQ(e.capacity(), 30);
}

TEST(EventTest, RejectsCapacityBelowOne) {
    // extra parentheses: the macro sees one argument
    EXPECT_THROW((Event{2, "Talk", kStart, "Hall", 0}),
                 std::invalid_argument);
}

TEST(EventTest, HasStartedFromItsStartTime) {
    const Event e{3, "Hackathon", kStart, "Lab 1", 40};
    EXPECT_FALSE(e.hasStarted(kStart - 1min));
    EXPECT_TRUE(e.hasStarted(kStart));
}
```

```cmake
# CMakeLists.txt (from Lab-08, test part)
include(FetchContent)
FetchContent_Declare(googletest
  URL https://github.com/google/googletest/archive/v1.15.2.zip)
FetchContent_MakeAvailable(googletest)

enable_testing()
add_executable(clubhub_tests
  tests/event_test.cpp tests/registration_service_test.cpp)
target_link_libraries(clubhub_tests
  PRIVATE clubhub_core GTest::gtest_main)
include(GoogleTest)
gtest_discover_tests(clubhub_tests)
```

Macro	Checks
`EXPECT_EQ(a, b)`, `EXPECT_NE`, `EXPECT_LT`	comparison; on failure prints both values
`EXPECT_TRUE(c)`, `EXPECT_FALSE(c)`	a condition
`EXPECT_THROW(stmt, Type)`, `EXPECT_NO_THROW`	the statement throws that exception type (or nothing)
`ASSERT_*` versions of all of the above	same check, but a failure **stops** the test

Structure each test as **Arrange, Act, Assert**. Use `ASSERT_` when the rest of the test makes no sense after a failure (for example before dereferencing an `std::optional`), `EXPECT_` otherwise so that one run reports every failed check.

A fixture with `TEST_F`, and time you control instead of `system_clock::now()`

```cpp
// include/clubhub/clock.hpp
using TimePoint =
    std::chrono::system_clock::time_point;

class Clock {
public:
    virtual ~Clock() = default;
    virtual TimePoint now() const = 0;
};

// production: RegistrationService gets this
class SystemClock final : public Clock {
public:
    TimePoint now() const override {
        return std::chrono::system_clock::now();
    }
};

// tests/fake_clock.hpp: time stands still
class FakeClock final : public Clock {
public:
    explicit FakeClock(TimePoint t) : now_{t} {}
    TimePoint now() const override { return now_; }
    void set(TimePoint t) { now_ = t; }
private:
    TimePoint now_;
};
```

⚠ A service that calls `system_clock::now()` itself gives tests that pass today and fail next month, when the hard-coded event date lies in the past. Injecting the clock makes time a test input.

```cpp
// tests/registration_service_test.cpp
using namespace std::chrono;
using namespace clubhub;

class RegistrationServiceTest : public ::testing::Test {
protected:
    static constexpr int kEventId = 1;
    const TimePoint start = sys_days{2026y / November / 20} + 18h;
    // three days before the start
    FakeClock clock{start - 72h};
    InMemoryEventRepository events;
    InMemoryRegistrationRepository regs;
    RegistrationService service{events, regs, clock};

    // runs before every TEST_F: a fresh event with 2 seats
    void SetUp() override {
        events.save(Event{kEventId, "C++ Workshop", start, "B-201", 2});
    }
};

TEST_F(RegistrationServiceTest, RegistersWhenSeatIsFree) {
    service.registerStudent("e20230001", kEventId);
    const auto reg = regs.find("e20230001", kEventId);
    // stop here if missing: reg-> would be undefined behaviour
    ASSERT_TRUE(reg.has_value());
    EXPECT_EQ(reg->status, RegistrationStatus::Registered);
}

TEST_F(RegistrationServiceTest, RejectsEventThatHasStarted) {
    clock.set(start);
    EXPECT_THROW(service.registerStudent("e20230002", kEventId),
                 RuleViolation);
}
```

Parameterised tests: one test body, every boundary value

```
struct CancelCase {
    minutes beforeStart;
    bool allowed;
};
// readable parameter values in gtest and ctest output
void PrintTo(const CancelCase& c, std::ostream* os) {
    *os << c.beforeStart.count() << "min_before_"
        << (c.allowed ? "allowed" : "rejected");
}

// reuse the fixture, add a parameter
class CancelDeadlineTest
    : public RegistrationServiceTest,
      public ::testing::WithParamInterface<CancelCase> {};

TEST_P(CancelDeadlineTest, AppliesThe24HourRule) {
    const auto [beforeStart, allowed] = GetParam();
    service.registerStudent("e20230001", kEventId);
    clock.set(start - beforeStart);
    if (allowed) {
        EXPECT_NO_THROW(service.cancel("e20230001", kEventId));
    } else {
        EXPECT_THROW(service.cancel("e20230001", kEventId),
                     RuleViolation);
    }
}

// D - 1 min, D, D + 1 min, and the start itself
INSTANTIATE_TEST_SUITE_P(Boundaries, CancelDeadlineTest,
    ::testing::Values(CancelCase{24h + 1min, true},
                      CancelCase{24h, true},
                      CancelCase{24h - 1min, false},
                      CancelCase{0min, false}));
```

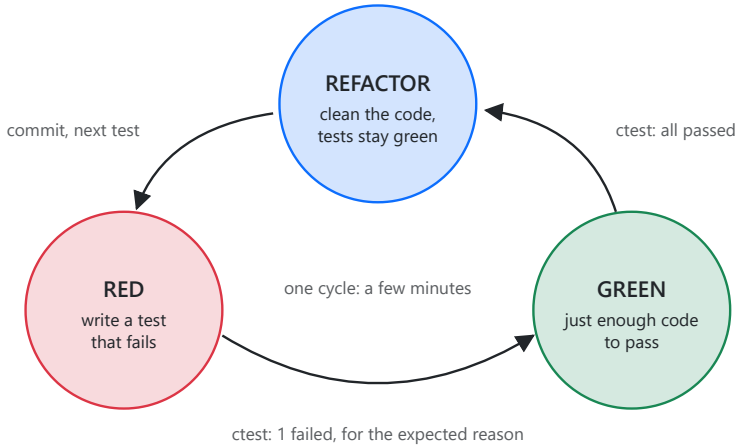
Running the suite against the faulty `cancel` of slide 5:

```
[ RUN      ] Boundaries/CancelDeadlineTest.
             AppliesThe24HourRule/2
registration_service_test.cpp:63: Failure
Expected: service.cancel("e20230001", kEventId)
  throws an exception of type RuleViolation.
  Actual: it throws nothing.
[  FAILED  ] ...AppliesThe24HourRule/2,
  where GetParam() = 1439min_before_rejected
[ RUN      ] ...AppliesThe24HourRule/3
  (same failure)
[  FAILED  ] ...AppliesThe24HourRule/3,
  where GetParam() = 0min_before_rejected
[  PASSED  ] 2 tests.
[  FAILED  ] 2 tests
```

- Cases `/2` (D + 1 min) and `/3` (the start) fail; both cases before the deadline pass. A test at 72 h before the start would never have noticed.
- Each value is reported separately; `PrintTo` makes the failure name the boundary that broke.
- The same pattern covers capacity (k = 28, 29, 30) and check-in windows.

The fixture members (`service`, `clock`, `start`) are inherited, so the parameterised test reuses the whole Arrange step.

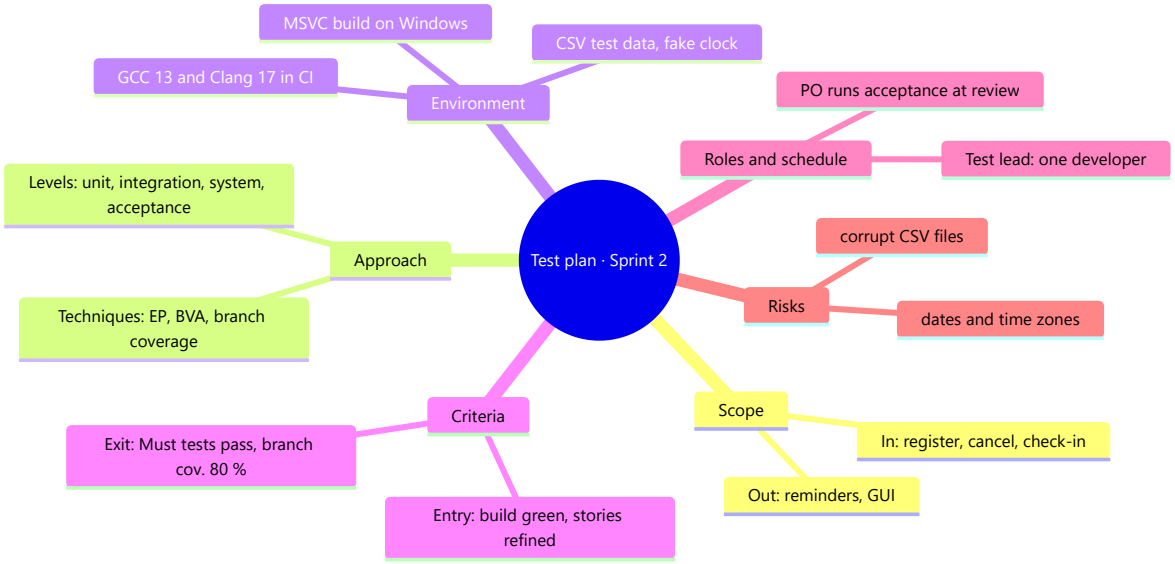
TDD: write the failing test first, then the code, then clean up




```
// RED: check-in only for registered students
TEST_F(RegistrationServiceTest, CheckInNeedsRegistration) {
    clock.set(start - 10min);
    EXPECT_THROW(service.checkIn("e20230009", kEventId),
                 RuleViolation);
}

// GREEN: the smallest change that passes
void RegistrationService::checkIn(
        const std::string& studentId, int eventId) {
    const auto reg = regs_.find(studentId, eventId);
    if (!reg || reg->status != RegistrationStatus::Registered) {
        throw RuleViolation{Rule::NotRegistered};
    }
    regs_.setStatus(studentId, eventId,
                    RegistrationStatus::CheckedIn);
}
// REFACTOR: cancel() has the same lookup; extract
// findActive(studentId, eventId), re-run ctest
```

TDD is an Extreme Programming practice (Chapter 07; Beck, *Test-Driven Development: By Example*, 2002). Its main benefit is design: code written to be tested first has injected dependencies, small functions and no hidden clock.

A test plan answers what, how, where, who, and when we are done



- The **test plan** is short in an agile team: one page per sprint, reviewed at sprint planning.
- **Entry criteria** say when testing can start; **exit criteria** say when it is finished. Both must be measurable.
- *Weak*: "test thoroughly". *Strong*: "every Must test case passes; branch coverage of `RegistrationService` is at least 80 %; no open Critical or Major defect".
- The plan names **risks** so that test effort goes where failures would hurt most (risk-based testing).

Structure inspired by ISO/IEC/IEEE 29119-3:2021 (test documentation), trimmed to what a student team needs.

System-level test cases for the CLI, and the report at the end of the sprint

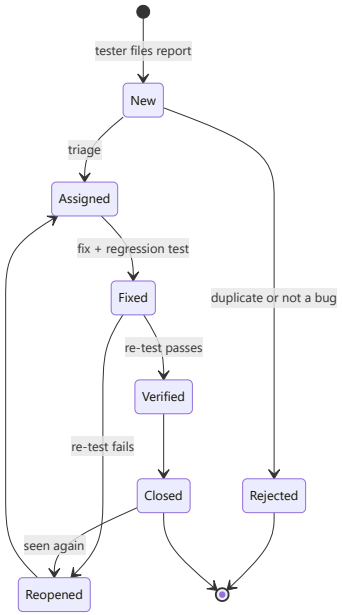
Id	Precondition	Steps	Input	Expected	FR
TC-REG-01	Event 1 "C++ Workshop", C = 30, 12 registered; now = S − 72 h	1. `clubhub register` 2. `clubhub list-registrations 1`	student e20230001, event 1	"Registered." exit code 0; 13 rows with status Registered	FR-11
TC-REG-03	Event 1 full: 30 registered	1. `clubhub register`	e20230031, event 1	"Error: event is full" exit code 2; still 30 rows	FR-11
TC-CAN-02	e20230001 registered; now = S − 24 h	1. `clubhub cancel`	e20230001, event 1	"Cancelled." status Cancelled; 29 seats taken	FR-12
TC-CAN-03	e20230001 registered; now = S − 23 h 59 min	1. `clubhub cancel`	e20230001, event 1	"Error: cancellation closed 24 h before start" exit code 2; still Registered	FR-12
TC-CHK-01	e20230001 registered; now = S − 10 min	1. `clubhub check-in` 2. `clubhub attendance 1`	e20230001, event 1	Status CheckedIn; attendance shows 1 / 30	FR-13

A good test case is repeatable by someone else: exact preconditions (including the time), exact input, one observable expected result, and the requirement it verifies.

```
Test summary report
Sprint 2 · 2026-11-27 · v1.0

Executed 46   Passed 43
Failed    2   Blocked 1
Line coverage   89.0 %
Branch coverage 81.3 %
  (RegistrationService)

Failed: TC-CAN-03 -> bug #47
        TC-CHK-04 -> bug #52
Blocked: TC-REM-01 (reminder
  story not done)

Exit criteria: met except
#47 (Major, fix in review)
Recommendation: release after
#47 is verified
```

The **test report** tells the Product Owner whether the increment is good enough to release, with evidence, not feelings.

Every defect has a state, an owner and a history



	Severity	Priority
Means	How bad is the impact?	How soon must it be fixed?
Set by	Tester	Product Owner
Scale	Critical, Major, Minor, Trivial	High, Medium, Low

- Typo in the club name on the demo screen the day before the demo: **Trivial** severity, **High** priority.
- Monthly report crashes when `events.csv` has an empty last line, but the report is only needed in week 14: **Critical** severity, **Medium** priority.
- On GitHub: states map to labels and the project board; "Fixed" means a merged PR whose description says `Fixes #47`.

A bug report is a request for someone else's time: make it reproducible

⊗ **Poorly written**

Title: cancel doesnt work!!!

I tried to cancel and it was wrong. It worked yesterday. Please fix ASAP.

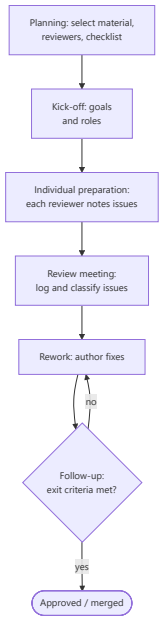
Missing	Why it matters
Steps and data	The developer cannot reproduce it
Expected vs actual	"Wrong" compared with what?
Version, environment	Which commit, which compiler?
Severity, requirement	No basis for triage
Neutral tone	Blame slows the fix down

One defect per report; a specific, searchable title; facts, not guesses (put guesses under "Notes").

```
### #47 cancel accepted 23 h before start (violates FR-12)

**Environment**: commit 4f2c9e1 · Windows 11 · MSVC 19.40 · Debug
**Severity**: Major · **Priority**: High · **Found by**: TC-CAN-03

**Preconditions**: event 1 "C++ Workshop" starts 2026-11-20 18:00;
student e20230001 is Registered.

**Steps to reproduce**
1. clubhub --now "2026-11-19 19:00" cancel e20230001 1
2. clubhub list-registrations 1

**Expected**: "Error: cancellation closed 24 h before start",
exit code 2, status stays Registered.
**Actual**: "Cancelled.", exit code 0, status Cancelled in
data/registrations.csv (excerpt attached).

**Notes**: 25 h before the start works correctly. The fault is
probably in the deadline comparison in RegistrationService::cancel.
```

The `--now` option exists because the CLI builds its `RegistrationService` with an injected clock: testability pays off in manual testing too.

Reviews find defects before anything runs, in code and in documents



Review type	Formality	Club Hub example
Informal / pair	none, immediate	Pair programming on `checkIn`
Pull request review	light: checklist, approval rule	Branch protection from Lab-08 requires one approval
Walkthrough	author leads the group	Author explains the CSV format to the team
Technical review	peers, documented result	Review of the class diagram before Sprint 2
Inspection	most formal: moderator, reader, scribe, metrics	Inspection of the SRS rules section

💡 Inspections were introduced by Fagan (IBM, 1976); IEEE 1028 describes the review types. Reviews also catch what tests cannot: unclear requirements, poor names, missing tests.

A code review checklist for C++, applied to a pull request

```cpp
// Under review: which checks fail?
class registration_service {
public:
    Registration* reg(std::string s, int e) {
        Event* ev = new Event(repo.get(e));
        if (ev->capacity() - count(e) < 1)
            return nullptr;
        try { save(s, e); } catch (...) {}
        return new Registration{s, e};
    }
    int count(int e) { return cnt[e]; }
    // ...
};
```

Review comment, good style: "Ownership (R.11): `new Event` leaks on every call. `repo.get(e)` already returns `const Event&`; can we use that reference directly?" Name the rule, the consequence and a suggestion; comment on code, never on people.

Area	Check	Found above	Guideline
Ownership	No naked `new`/`delete`; owners are values or `std::unique_ptr`; references for non-owning access	two `new`, two leaks, raw owning pointer returned	R.11, R.20
const	Queries are `const` member functions; large inputs by `const&`	`count` not const; `std::string s` copied	Con.2, F.16
Error handling	Broken rules throw `RuleViolation`; no empty `catch`; resources via RAII	`nullptr` as error signal; `catch (...) {}` hides save failures	E.2, E.6
Naming	`PascalCase` types, `camelCase` functions, `member_` suffix, no cryptic abbreviations	`registration_service`, `reg`, `s`, `e`, `cnt`	NL.8
Tests	Every new or changed rule has a test that fails without the change	no test in the PR	team DoD
Readability	Short functions, named constants (`kCancelWindow = 24h`), no dead code	magic `< 1` seat check	ES.45

Guideline ids refer to the C++ Core Guidelines (Stroustrup and Sutter, maintained online). Keep the checklist in `.github/pull_request_template.md` so every PR shows it.

Static analysis: tools review every line on every push

```
# compile_commands.json tells the tools how each file is built
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
clang-tidy -p build src/*.cpp
cppcheck --enable=warning,style,performance --std=c++20 \
         --project=build/compile_commands.json --error-exitcode=1
```

```
src/registration_service.cpp:18:43: warning: the parameter 'studentId'
    is copied for each invocation but only used as a const reference;
    consider making it a const reference [performance-unnecessary-value-param]
src/csv_event_repository.cpp:40:21: warning: narrowing conversion from
    'size_t' to signed type 'int' is implementation-defined
    [cppcoreguidelines-narrowing-conversions]
src/event.cpp:12:5: warning: Member variable 'Event::capacity_' is not
    initialized in the constructor. [uninitMemberVar]
src/report.cpp:27:10: style: Consider using std::count_if algorithm
    instead of a raw loop. [useStlAlgorithm]
```

```
# .clang-tidy (repository root)
Checks: >
  bugprone-*, cppcoreguidelines-*, modernize-*, performance-*,
  readability-*, -modernize-use-trailing-return-type
WarningsAsErrors: 'bugprone-*'
```

Finding	Fix
value parameter copied	`std::string studentId` → `const std::string& studentId`
narrowing `size_t` → `int`	return `std::size_t`, or `static_cast<int>` after checking the range
uninitialised member	default member initialiser `int capacity_{0};` and validate in the constructor
raw loop	`std::ranges::count_if(regs, isCheckedIn)`

- **clang-tidy** checks style, modern C++ and the C++ Core Guidelines; **cppcheck** looks for undefined behaviour and bugs with few false positives.
- Compiler warnings are the cheapest static analysis: build with `-Wall -Wextra -Wpedantic` (GCC, Clang) or `/W4` (MSVC).
- A false positive is suppressed *with a reason* (`// NOLINT(check): why`), never silently.

A **coding standard** (C++ Core Guidelines plus `.clang-format`) makes reviews about design instead of brace placement.

ISO/IEC 25010:2023: nine product quality characteristics

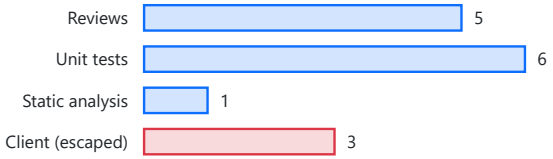
Characteristic	Measurable target for Club Hub
Functional correctness	all Must rule tests pass in CI
Time behaviour	`list-events` with 5000 events under 1 s
Reliability (recoverability)	a corrupt CSV line is reported and skipped, no crash
Security (confidentiality)	no phone numbers in the attendance export
Maintainability (testability)	branch coverage of rules ≥ 80 %, clang-tidy clean
Interaction capability	every error message names the rule and what to do

The 2023 edition renamed usability to *interaction capability* and portability to *flexibility*, and added *safety*. A quality model turns "good software" into characteristics you can specify and measure (Chapter 02 used it for quality attributes).

Quality metrics: defect density, coverage and escaped defects

```
ITC Club Hub after Sprint 2
  C++ core size (non-blank, non-comment)    2 400 lines = 2.4 KLOC
  Defects found by the team                12
    reviews 5, unit tests 6, static analysis 1
  Defects found by the client afterwards    3  (escaped)

Defect density   = (12 + 3) / 2.4 KLOC  = 6.25 defects per KLOC
Escaped defects  = 3 / (12 + 3)         = 20 %
Detection effectiveness = 12 / 15       = 80 %
Branch coverage  = 50 / 68 outcomes     = 73.5 %
```

Where the 15 defects were found



Metric	Definition	Use it to…
Defect density	defects / size (KLOC or story points)	compare modules: which one needs a review or a rewrite?
Coverage	covered statements or branches / total	find untested code; not a proof of quality
Escaped defects	defects found after release / all defects	judge the whole QA process; aim to push it down each sprint
Test pass rate	passed / executed test cases	decide on release together with open severities

⚠ When a measure becomes a target, it stops being a good measure (Goodhart's law): a coverage target without review produces tests that assert nothing. Use metrics to ask questions, not to rank people. Here the 3 escaped defects were all in check-in, the story with no boundary tests.

Best practices and common mistakes in testing and QA

1 Design tests from the rules

Partitions and boundaries first, code second. Each test case traces to an FR id, so a failing test tells the Product Owner which promise is broken.

2 Make time and files inputs

Inject the clock and the repositories. Tests that read the real clock or a shared `data/` folder are flaky and order-dependent.

3 One behaviour per test, clear names

`RejectsSecondRegistration` tells you what broke without opening the file. Arrange, Act, Assert; no logic or loops inside the test body.

4 Every bug gets a test first

Write the test that reproduces bug #47, watch it fail, then fix. The test stays as a regression guard forever.

5 Let machines do the boring review

Warnings, clang-tidy, cppcheck and clang-format in CI; human reviewers spend their time on design, rules and tests.

6 Measurable exit criteria

"Must tests pass, branch coverage ≥ 80 % on the rules, no open Major" can be checked by anyone. "Tested enough" cannot.

Mistake: tests without assertions

Calling `registerStudent` and checking nothing raises coverage and finds no bug. Coverage is a flashlight, not a grade.

Mistake: only the happy path

Most faults live in invalid partitions and at boundaries: full events, the 24-hour edge, a second registration, a corrupt CSV line.

Mistake: testing at the end

"We test in week 13" means defects are found when fixing them is most expensive. Test inside each sprint, as part of the definition of done.

Chapter quiz 10 questions

1. A developer writes `if (now > start)` instead of `if (now > start - 24h)`. What is that wrong line?

- An error
- A fault
- A failure
- A validation issue

2. The team demonstrates the CLI to the club leader to check that it really solves the club's problem. This activity is...

- Verification
- Static analysis
- Validation
- Regression testing

3. A test runs `RegistrationService` together with `CsvRegistrationRepository` writing a real temporary file. Which level is it?

- Unit
- Integration
- System
- Acceptance

4. Capacity is 30 and k is the number of seats already taken. Which set gives one representative for each valid partition, "seat free" and "full"?

- k = 5 and k = 40
- k = 5 and k = 10
- k = 28 and k = 29
- k = 40 and k = 50

5. Cancellation is allowed up to and including 24 h before the start S. Which values does 2-value boundary value analysis test for this rule (1-minute resolution)?

- S − 48 h and S
- S − 24 h and S − 24 h + 1 min
- S − 25 h and S − 23 h
- Only S − 24 h

6. For `admit(taken, capacity, waitlistOn)` with two `if` decisions, the tests (10, 30, false) and (30, 30, true) are run. What is the branch coverage?

- 25 %
- 50 %
- 75 %
- 100 %

7. Why does the test call `ASSERT_TRUE(reg.has_value())` before `EXPECT_EQ(reg->status, …)`?

- ASSERT is faster than EXPECT
- To stop the test before dereferencing an empty optional
- EXPECT_EQ cannot compare enums
- ASSERT_TRUE also checks the status

8. Why does `RegistrationService` receive a `Clock` instead of calling `system_clock::now()`?

- It makes the program run faster
- Tests can set the time exactly, so time rules are deterministic
- GoogleTest cannot link against <chrono>
- The C++ standard forbids calling now() in a class

9. A bug is in state Fixed. The tester re-runs the test and it still fails. Which state comes next?

- Closed
- Rejected
- Verified
- Reopened

10. The team found 12 defects before release and the client found 3 afterwards. What share of defects escaped?

- 3 %
- 20 %
- 25 %
- 80 %

WRAP-UP

Summary

Verification checks the product against its specification; validation checks it against the client's real needs.	Equivalence partitioning picks one test per class; boundary value analysis tests both sides of every edge.	A test plan with measurable exit criteria, repeatable test cases and a sprint test report guide the release decision.
An error (human) leaves a fault (code) that may cause a failure (run time) only with a triggering input.	Branch coverage is stronger than statement coverage; gcov and gcovr show which lines and outcomes were never run.	Good bug reports are reproducible and move through New, Assigned, Fixed, Verified, Closed.
Unit, integration, system and acceptance tests form a pyramid: many fast unit tests, few slow end-to-end ones.	GoogleTest fixtures, `TEST_P` boundaries and an injected clock make time rules testable; TDD writes the test first.	Reviews, static analysis and the C++ Core Guidelines prevent defects; ISO/IEC 25010 and metrics make quality measurable.

Open Lab-09: 5 tasks + 1 challenge

Next chapter: 10 · Software Maintenance and Evolution