https://simplifycpp.org

Testable C++

Writing Code That Can Be Easily Tested

Prepared by Ayman alheraki

Testable C++

Writing Code That Can Be Easily Tested

Prepared by Ayman Alheraki

simplifycpp.org

December 2025

Contents

Author's Introduction

This booklet was written for professional C++ engineers working on large, long-lived software systems—systems measured not in weeks or months, but in years and decades—where architectural mistakes are far more expensive than implementation bugs.

In such systems, a bug is rarely catastrophic on its own. Bugs can be fixed. What cannot be fixed easily is a design that resists understanding, verification, and change.

Over years of maintaining real-world C++ systems, one lesson becomes unavoidable: most failures do not originate from complex algorithms or advanced language features. They originate from code that cannot be tested safely, reliably, or in isolation.

When code is difficult to test, engineers avoid changing it. When engineers avoid changing code, systems stagnate. When systems stagnate, technical debt accumulates silently until even minor modifications become risky.

At that point, the system is no longer controlled by its designers. It is controlled by fear.

Testability is often misunderstood as a concern of tooling. Teams debate testing frameworks, mocking libraries, or coverage tools, while ignoring the far more fundamental issue: whether the code was designed to be testable in the first place.

This booklet takes a different position.

Testability is not a testing-framework problem. It is not a tooling concern, It is not something that can be added after the fact, Testability is a design property.

It emerges naturally when software is built with clear boundaries, explicit dependencies, deterministic behavior, and controlled state. It disappears when construction logic is hidden,

ownership is implicit, and side effects are scattered across the codebase.

C++ occupies a unique position among programming languages. It offers unmatched control over object construction, lifetime, resource management, and performance. That control is the foundation of C++'s continued relevance in systems programming, high-performance computing, embedded systems, and large industrial applications.

However, that same control can either enable architectural clarity or silently destroy it.

In C++, nothing prevents an engineer from constructing dependencies deep inside functions, relying on global state, hiding ownership rules, or coupling components tightly through implicit assumptions. The language allows it—and will often compile it efficiently.

The cost of such designs is not paid at compile time. It is paid years later, during maintenance, testing, and refactoring.

This booklet treats testability as an architectural discipline rooted in core C++ principles rather than borrowed patterns from managed languages.

It does not attempt to turn C++ into Java, C#, or any framework-driven ecosystem. It does not advocate excessive abstraction, overuse of interfaces, or mocking everything that moves.

Instead, it embraces what C++ does best: explicit construction, deterministic lifetime, clear ownership, and precise dependency control.

When these principles are applied consistently, testability is no longer an external requirement imposed by process or policy. It becomes a natural consequence of good design.

Well-designed C++ code tends to be testable for the same reasons it is maintainable: it is understandable, modular, and honest about its dependencies.

The goal of this booklet is not to teach how to write tests. It is to teach how to write code that *wants* to be tested.

Code that can be tested easily is code that can be reasoned about. Code that can be reasoned about can be refactored safely. Code that can be refactored safely can evolve.

And code that can evolve is code that survives change.

Ayman Alheraki

Preface

In professional software systems, testing is not optional. It is not a luxury, and it is not a process checkbox. Testing is the only scalable mechanism for preserving correctness as software evolves under continuous change.

Unlike documentation or informal review, tests execute. They validate assumptions repeatedly, under automation, and at scale. They provide the only practical feedback loop capable of detecting regressions before they reach production systems.

Yet despite this reality, many C++ codebases actively resist testing.

The resistance does not come from the language itself. It comes from architectural decisions that make software difficult to observe, isolate, and control.

Hidden dependencies constructed deep inside functions, reliance on global and static state, implicit initialization order, and tightly coupled components all conspire to make even simple tests fragile and expensive to write.

In such environments, tests break for reasons unrelated to behavior. Engineers lose trust in the test suite. Over time, tests are avoided rather than expanded, and coverage becomes superficial.

The result is predictable and repeatable across organizations: regressions accumulate silently, refactoring becomes dangerous, and changes are delayed or abandoned out of caution.

Eventually, teams begin to treat the existing codebase as untouchable. The system continues to function, but it no longer evolves with confidence.

This booklet is built around a single, uncompromising idea: *testability must be designed, not*

added later.

No testing framework, no mocking library, and no coverage tool can compensate for code that was not designed to be testable. Attempts to retrofit tests onto such systems often result in brittle, over-mocked test suites that mirror the complexity of the production code without providing real assurance.

Testability is not achieved by writing more tests. It is achieved by reducing architectural friction.

When code is constructed deliberately, dependencies are explicit rather than hidden. When ownership and lifetime are clear, behavior becomes predictable. When state is controlled and localized, outcomes can be observed and verified.

Under these conditions, testing stops being an obstacle. It becomes a natural extension of design.

This booklet does not promote testing for its own sake. It promotes clarity, discipline, and architectural honesty.

By focusing on how C++ code is structured, constructed, and connected, the techniques presented here aim to make testing a consequence of good design rather than a continuous struggle against it.

When dependencies are explicit and state is controlled, testing becomes simpler, faster, and more reliable. And when testing becomes reliable, change becomes safe.

That is the foundation upon which long-lived C++ systems are built.

Chapter 1

What Makes Code Testable

Testable code is not defined by the presence of tests. It is defined by the absence of resistance to testing.
When code is designed correctly, tests become straightforward, predictable, and stable. When code is designed poorly, tests become fragile, expensive, and misleading.
Over time, experienced engineers recognize that testability is not an emergent property of code volume or developer discipline. It is the direct result of a small number of architectural properties applied consistently across the system.
Testable code exhibits a small number of recurring characteristics:

- Explicit dependencies
- Deterministic behavior
- Observable outcomes
- Minimal hidden state

These properties are independent of testing frameworks. They arise from architectural discipline rather than tooling choices.

This chapter examines each property in detail and demonstrates how seemingly minor design decisions can either enable or destroy testability in C++ systems.

1.1 Explicit Dependencies

A dependency is any external component required for a unit of code to perform its work. Filesystems, databases, clocks, loggers, network clients, and random number generators are all dependencies.
Code with implicit dependencies hides its requirements. Code with explicit dependencies declares them.
Hidden dependencies make testing difficult because they cannot be replaced, controlled, or observed.

1.1.1 Hidden Dependency: Untestable Design

```
class FileLogger {
public:
    void write(const std::string& msg);
};

class Processor {
public:
    void run() {
        FileLogger logger;
        logger.write("Processing started");
    }
};
```

In this design, the dependency is constructed internally. A test has no way to:

- Observe what was written

- Prevent file I/O
- Replace the logger with a test double

The only way to test this code is indirectly, often through the filesystem, which makes tests slow and fragile.

1.1.2 Explicit Dependency: Testable Design

```
class Logger {
public:
    virtual ~Logger() = default;
    virtual void write(const std::string& msg) = 0;
};

class Processor {
public:
    explicit Processor(Logger& logger)
        : logger_(logger) {}

    void run() {
        logger_.write("Processing started");
    }

private:
    Logger& logger_;
};
```

The behavior is unchanged, but the architecture is transformed. Tests can now provide a fake or mock logger and observe behavior directly.
Explicit dependencies reduce coupling and increase clarity.

1.2 Deterministic Behavior

A test must produce the same result every time it is executed. Any source of nondeterminism undermines confidence in the test suite.
Common sources of nondeterminism include:

- Current time
- Random numbers
- Global mutable state
- Uncontrolled concurrency

1.2.1 Nondeterministic Code

```
#include <ctime>

bool isExpired(std::time_t expiry) {
    return std::time(nullptr) > expiry;
}
```

This function cannot be tested reliably because the result depends on the wall clock.

1.2.2 Deterministic Design Through Abstraction

```
#include <ctime>

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

```
bool isExpired(const Clock& clock, std::time_t expiry) {
    return clock.now() > expiry;
}
```

By externalizing time, the function becomes deterministic. Tests can control time explicitly. Determinism is not about removing complexity. It is about controlling it.

1.3 Observable Outcomes

Code that performs work without exposing results cannot be tested meaningfully.
Side effects that disappear into files, logs, or network sockets without observation force tests to rely on indirect verification.

1.3.1 Unobservable Behavior

```
void processData() {
    // performs work
    // writes results internally
}
```

This function provides no observable outcome. A test cannot verify correctness without inspecting internal details or external side effects.

1.3.2 Observable Outcomes Through Return Values

```
enum class Result {
    Success,
    Failure
};
```

```
Result processData() {
    // perform work
    return Result::Success;
}
```

Explicit outcomes allow tests to assert behavior directly.

1.3.3 Observable Outcomes Through State

```
class Processor {
public:
    void run() {
        processed_ = true;
    }

    bool processed() const {
        return processed_;
    }

private:
    bool processed_ = false;
};
```

Well-defined observable state makes behavior verifiable without breaking encapsulation.

1.4 Minimal Hidden State

Hidden state introduces implicit behavior that depends on execution order and history.
Global variables and static initialization are the most common sources of hidden state in C++ systems.

1.4.1 Global State: Fragile Tests

```
int counter = 0;

void increment() {
    ++counter;
}
```

Tests that depend on this function must now coordinate execution order and reset state manually.

1.4.2 Localized State: Predictable Design

```
class Counter {
public:
    void increment() {
        ++value_;
    }

    int value() const {
        return value_;
    }

private:
    int value_ = 0;
};
```

By localizing state, behavior becomes predictable and test isolation is restored.
Hidden state increases cognitive load and erodes trust in tests. Minimal state reduces both.

1.5 Testability as a System Property

Testability is not achieved by fixing individual functions. It is a system-wide property that emerges when consistent architectural rules are applied.
A single hidden dependency can infect an entire subsystem. A single global variable can destabilize dozens of tests.
Conversely, systems built around explicit construction, controlled state, and observable behavior naturally support testing at all levels: unit, integration, and system tests.
Testable code is not necessarily simpler. It is clearer.
And clarity, more than cleverness, is the foundation of long-lived software.

Chapter 2

Construction Is Architecture

In C++, object construction is not a mechanical detail. It is architecture.

Construction determines ownership, lifetime, coupling, and visibility. It defines which components are responsible for creating other components, which components can be replaced, and which components are permanently bound together.

In small programs, construction choices may appear harmless. In large, long-lived systems, they determine whether a codebase remains flexible or becomes rigid over time.

Where an object is constructed often matters more than how it is implemented.

Hidden construction hard-wires behavior, conceals dependencies, and blocks substitution during tests. Once construction logic is buried inside functions or classes, tests lose the ability to control behavior without invasive changes.

This chapter explores why construction is a first-class architectural decision in C++, and how deliberate construction enables testability without sacrificing performance or clarity.

2.1 Hidden Construction and Its Consequences

Hidden construction occurs when a component creates its own dependencies internally rather than receiving them from the outside.
At first glance, this may appear convenient. In reality, it couples the component to specific implementations and execution contexts.

2.1.1 Hidden Dependency: Rigid Design

```
class Logger {
public:
    void log(const std::string& msg);
};

class Service {
public:
    void execute() {
        Logger logger;            // Hidden construction
        logger.log("Running");
    }
};
```

In this design:

- The service is permanently bound to a concrete logger
- File or console I/O may occur during tests
- Behavior cannot be observed without side effects

A test has no way to observe, replace, or suppress this dependency. The only way to test the service is indirectly, often through external effects such as files or output streams.

2.1.2 The Illusion of Simplicity

Hidden construction feels simple because it reduces parameter lists and appears self-contained. However, this simplicity is deceptive.
The complexity has not been removed. It has been hidden.
As systems grow, hidden construction spreads across the codebase, creating a network of tightly coupled components that cannot be tested or evolved independently.

2.2 Explicit Construction and Dependency Injection

Explicit construction means that a component does not create its own dependencies. Instead, dependencies are provided from the outside.
This approach is often referred to as dependency injection, but in C++ it requires no frameworks, containers, or reflection.
It is simply a matter of constructor design.

2.2.1 Explicit Dependency: Flexible Design

```
class Logger {
public:
    virtual ~Logger() = default;
    virtual void log(const std::string& msg) = 0;
};

class Service {
public:
    explicit Service(Logger& logger)
        : logger_(logger) {}

    void execute() {
```

```
        logger_.log("Running");
    }

private:
    Logger& logger_;
};
```

With this design:

- The service declares its dependency explicitly
- Tests can substitute the logger
- Behavior can be observed without side effects

The functionality is unchanged. The architecture is transformed.
This single change converts rigid code into testable code.

2.3 Construction and Ownership Semantics

In C++, construction is inseparable from ownership and lifetime. A constructor communicates whether a component:

- Owns a dependency
- Borrows a dependency
- Shares ownership

Ambiguous ownership leads to resource leaks, dangling references, and confusion during testing.

2.3.1 Ambiguous Ownership

```
class Service {
public:
    Service(Logger* logger)
        : logger_(logger) {}

private:
    Logger* logger_;
};
```

This design raises unanswered questions:

- Who owns the logger?
- Who deletes it?
- Can it be null?

Such ambiguity complicates both production code and tests.

2.3.2 Clear Ownership Through References

```
class Service {
public:
    explicit Service(Logger& logger)
        : logger_(logger) {}

private:
    Logger& logger_;
};
```

A reference communicates non-ownership and non-null semantics clearly.

2.3.3 Ownership Through Smart Pointers

```
#include <memory>

class Service {
public:
    explicit Service(std::unique_ptr<Logger> logger)
        : logger_(std::move(logger)) {}

private:
    std::unique_ptr<Logger> logger_;
};
```

Ownership is now explicit and enforced by the type system.
Tests can construct services with test-specific dependencies without confusion or leaks.

2.4 Factories and Testability

Factories centralize construction logic. Used carefully, they can improve testability. Used carelessly, they reintroduce hidden construction.

2.4.1 Hard-Wired Factory

```
class ServiceFactory {
public:
    static Service create() {
        static FileLogger logger;
        return Service(logger);
    }
};
```

This factory permanently binds the service to a specific implementation.

2.4.2 Injectable Factory

```
class ServiceFactory {
public:
    explicit ServiceFactory(Logger& logger)
        : logger_(logger) {}

    Service create() const {
        return Service(logger_);
    }

private:
    Logger& logger_;
};
```

Factories themselves must follow the same testability rules as any other component.

2.5 Construction as a Boundary

In testable systems, construction occurs at boundaries:

- Application startup
- Composition roots
- High-level orchestration layers

Business logic should rarely construct infrastructure components directly.
By pushing construction outward, systems gain:

- Clear dependency graphs
- Easier substitution in tests

- Improved reasoning about lifetimes

Construction is not an implementation detail. It is a statement of architectural intent.

2.6 Why Construction Determines Testability

Tests require control. Control requires explicit construction.
When components construct their own dependencies, control is lost. When dependencies are provided deliberately, control is restored.
This is why construction choices determine whether code can be tested without friction.
In C++, architecture is not defined by diagrams or documentation. It is defined by constructors.
And constructors, more than any other mechanism, determine whether a system remains flexible or becomes rigid over time.

Chapter 3

Interfaces Without Abuse

Interfaces are one of the most powerful tools available to C++ engineers—and one of the most frequently misused.
Used correctly, interfaces isolate volatility, clarify intent, and enable substitution. Used incorrectly, they introduce indirection, cognitive overhead, and artificial complexity without improving design.
This chapter argues for a disciplined, purpose-driven use of interfaces in C++, rooted in architectural necessity rather than pattern worship.
Interfaces do not exist to satisfy design patterns. They exist to manage change.

3.1 The Purpose of an Interface

An interface defines a boundary between stable code and volatile code. It exists where change is expected, frequent, or outside the control of the system.
Examples of inherently volatile concepts include:

- Time

- Randomness
- I/O
- External services
- Operating system facilities

Interfaces are justified at these boundaries because they protect the core logic from instability. Introducing interfaces elsewhere often adds cost without benefit.

3.2 The Cost of Over-Abstraction

Over-abstraction occurs when interfaces are introduced preemptively, without evidence of volatility or need for substitution.

3.2.1 Unnecessary Interface

```
class Calculator {
public:
    virtual ~Calculator() = default;
    virtual int add(int a, int b) const = 0;
};
```

This interface provides no benefit. Addition is deterministic, stable, and trivial. Abstracting it does not improve testability or flexibility.
Instead, it increases:

- Indirection
- Boilerplate
- Cognitive load

Tests do not become clearer. They become more verbose.

3.2.2 Concrete Code Is Often More Testable

```
int add(int a, int b) {
    return a + b;
}
```

This function is already perfectly testable. No abstraction improves it.
Interfaces should never be introduced to compensate for poor design elsewhere.

3.3 Under-Abstraction and Hidden Volatility

While over-abstraction is wasteful, under-abstraction is dangerous.
Failing to abstract volatile concepts binds logic to environmental details and destroys testability.
Time is a classic example.

3.3.1 Hard-Coded Time Dependency

```
#include <chrono>

bool isExpired(std::chrono::system_clock::time_point expiry) {
    return std::chrono::system_clock::now() > expiry;
}
```

This function is untestable without manipulating the system clock or introducing timing delays. Tests become slow, fragile, and nondeterministic.
The problem is not the function. The problem is the hidden dependency on time.

3.4 Abstracting Volatile Concepts

Abstracting time isolates nondeterminism and restores control.

3.4.1 Clock Interface

```
#include <chrono>

class Clock {
public:
    virtual ~Clock() = default;
    virtual std::chrono::system_clock::time_point now() const = 0;
};
```

This interface represents volatility explicitly.

3.4.2 Using the Interface

```
bool isExpired(const Clock& clock,
               std::chrono::system_clock::time_point expiry) {
    return clock.now() > expiry;
}
```

The function is now deterministic with respect to its inputs. Tests can control time precisely. This is not abstraction for abstraction's sake. It is abstraction driven by volatility.

3.5 Interfaces as Behavioral Contracts

An interface is a contract, not a collection of methods.
A well-designed interface:

- Is small
- Has a clear responsibility
- Expresses intent

3.5.1 Bloated Interface

```
class SystemServices {
public:
    virtual ~SystemServices() = default;
    virtual void log(const std::string&) = 0;
    virtual std::string hostname() const = 0;
    virtual int cpuCount() const = 0;
    virtual void sleepMs(int) = 0;
};
```

This interface mixes unrelated responsibilities. Tests must implement unnecessary behavior. Change becomes expensive.

3.5.2 Focused Interfaces

```
class Logger {
public:
    virtual ~Logger() = default;
    virtual void log(const std::string&) = 0;
};

class SystemInfo {
public:
    virtual ~SystemInfo() = default;
    virtual int cpuCount() const = 0;
};
```

Focused interfaces reduce coupling and improve clarity.

3.6 Interfaces and Test Doubles

Interfaces enable substitution, but substitution must be intentional.

3.6.1 Simple Test Double

```
class FakeClock : public Clock {
public:
    explicit FakeClock(std::chrono::system_clock::time_point now)
        : now_(now) {}

    std::chrono::system_clock::time_point now() const override {
        return now_;
    }

private:
    std::chrono::system_clock::time_point now_;
};
```

The fake clock enables precise control without mocking frameworks.
This style scales better than heavy mocking and preserves clarity.

3.7 Prefer Interfaces at Boundaries

Interfaces belong at boundaries:

- Between business logic and infrastructure
- Between deterministic logic and nondeterministic inputs
- Between stable code and frequently changing code

They rarely belong inside pure algorithms or data structures.
When interfaces proliferate internally, design clarity is often decreasing rather than improving.

3.8 When Not to Use Interfaces

Do not introduce interfaces:

- To follow patterns blindly
- To “future-proof” without evidence
- To abstract simple, stable logic
- To compensate for poor construction choices

Interfaces are not free. They impose mental and structural cost.

3.9 Interfaces as Architectural Signals

A well-placed interface communicates architectural intent.
It signals:

- This part of the system may change
- This dependency must be controlled
- This boundary matters

An interface introduced without clear justification signals confusion.
In testable C++ systems, interfaces are rare, deliberate, and powerful.
They are not everywhere. They are exactly where they are needed—and nowhere else.

Chapter 4

Deterministic State

Tests require repeatability. Without repeatability, test results lose meaning and trust erodes. Deterministic code produces the same observable behavior every time it is executed with the same inputs. Non-deterministic code behaves differently depending on hidden factors such as execution order, timing, or shared state.

Global state is the most common source of non-determinism in C++ systems. It couples unrelated components, leaks behavior across test boundaries, and creates invisible dependencies that are difficult to reason about.

This chapter explores why deterministic state is essential for testable C++ and how to design stateful systems without sacrificing clarity or performance.

4.1 The Problem with Global State

Global variables introduce shared mutable state that persists across function calls, objects, and tests.

4.1.1 Global State Example

```
int globalCounter = 0;

void increment() {
    ++globalCounter;
}
```

At first glance, this code appears harmless. In practice, it introduces implicit coupling between every part of the system that touches the counter.
Tests that depend on this function now depend on execution order and hidden history.

4.1.2 Why Global State Breaks Tests

Global state:

- Persists across test cases
- Requires manual reset
- Couples unrelated tests
- Makes failures order-dependent

As test suites grow, these effects compound, leading to flakiness and false failures.

4.2 Hidden State and Temporal Coupling

Hidden state introduces temporal coupling: behavior depends on what happened before.

4.2.1 Temporal Coupling Example

```
void process() {
    increment();
    // behavior depends on previous calls
}
```

The correctness of `process` now depends on external history. Such code cannot be tested in isolation without reconstructing global context.
Temporal coupling increases cognitive load and makes reasoning about behavior difficult.

4.3 Localizing State

The primary strategy for restoring determinism is to localize state.
State should belong to objects with well-defined lifetimes, not to the program as a whole.

4.3.1 Localized State Through Objects

```
class Counter {
public:
    void increment() {
        ++value_;
    }

    int value() const {
        return value_;
    }

private:
    int value_ = 0;
};
```

Each instance now has its own state. Tests can construct fresh instances, ensuring isolation and repeatability.

4.3.2 Benefits of Localized State

Localized state:

- Eliminates test interference
- Clarifies ownership
- Makes lifetime explicit
- Restores determinism

4.4 Deterministic Initialization

Static initialization introduces subtle non-determinism due to unspecified initialization order across translation units.

4.4.1 Static Initialization Pitfall

```
static Logger globalLogger;

void logMessage(const std::string& msg) {
    globalLogger.log(msg);
}
```

The order in which `globalLogger` is initialized relative to other static objects is not guaranteed. Tests may fail depending on link order or build configuration.

4.4.2 Explicit Initialization

```
class Application {
public:
    explicit Application(Logger& logger)
        : logger_(logger) {}

    void log(const std::string& msg) {
        logger_.log(msg);
    }

private:
    Logger& logger_;
};
```

Explicit initialization removes uncertainty and restores control.

4.5 Controlling State in Tests

Testable systems allow tests to control state precisely.

4.5.1 State Injection

```
class Processor {
public:
    explicit Processor(int initialValue)
        : value_(initialValue) {}

    void increment() {
        ++value_;
    }
```

```
    int value() const {
        return value_;
    }

private:
    int value_;
};
```

Tests can now create processors in known states and verify behavior without side effects.

4.5.2 Reset Is a Code Smell

Code that requires explicit reset functions to be testable often hides state improperly.

```
void resetGlobalState();
```

Such functions indicate architectural issues rather than solutions.

4.6 Determinism and Concurrency

Concurrency introduces additional sources of non-determinism.
Shared mutable state across threads amplifies the problems of global state and temporal coupling.
While concurrency is unavoidable in many systems, deterministic design minimizes shared state and isolates synchronization points.

4.6.1 Shared State Across Threads

```
int sharedCounter = 0;
```

Without strict synchronization, behavior becomes unpredictable and untestable.
Deterministic concurrent systems confine shared state behind clear interfaces and well-defined synchronization strategies.

4.7 Deterministic State as an Architectural Goal

Deterministic state is not an optimization. It is a prerequisite for trust.

When state is localized, initialized explicitly, and owned clearly, tests become reliable and meaningful.

When state is global, implicit, or shared indiscriminately, tests become brittle and misleading.

Testable C++ systems do not eliminate state. They design it deliberately.

Determinism is not the absence of complexity. It is control over complexity.

Chapter 5

Error Handling and Testability

Error handling is one of the most overlooked dimensions of testability.
When failures are silent, ambiguous, or implicit, tests lose their ability to verify behavior.
When failures are explicit and observable, tests become precise and meaningful.
This chapter examines how different error-handling strategies in C++ affect testability, and how to design error reporting that supports clear verification without sacrificing performance or correctness.

5.1 Why Errors Must Be Observable

Tests exist to verify behavior under both normal and exceptional conditions. If errors cannot be observed, tests cannot assert correctness.
Silent failure forces tests to infer behavior indirectly, often through side effects or internal state inspection. Such tests are fragile and misleading.
Observable errors expose intent and allow tests to validate outcomes directly.

5.2 Silent Failure: A Design Smell

Silent failure occurs when a function fails without communicating that failure to the caller.

5.2.1 Silent Failure Example

```
int parseInt(const std::string& text) {
    try {
        return std::stoi(text);
    } catch (...) {
        return 0;   // Silent failure
    }
}
```

In this design, a failure is indistinguishable from valid input. Tests cannot determine whether parsing succeeded or failed.
This ambiguity forces callers to rely on conventions rather than guarantees.

5.3 Explicit Outcomes with `std::optional`

One of the simplest ways to make failure observable is to return an explicit success-or-failure type.

5.3.1 Using `std::optional`

```
#include <optional>
#include <string>

std::optional<int> parseInt(const std::string& text) {
    try {
        return std::stoi(text);
```

```
    } catch (...) {
        return std::nullopt;
    }
}
```

The return type now communicates intent clearly. Tests can verify both paths unambiguously.

5.3.2 Testability Benefits

Explicit outcomes:

- Eliminate ambiguity
- Simplify test assertions
- Clarify API contracts
- Reduce reliance on documentation

5.4 Exceptions and Testability

Exceptions are a powerful error-handling mechanism in C++. Used carefully, they improve testability. Used carelessly, they obscure control flow.

5.4.1 Exceptions as Signals

```
int parseInt(const std::string& text) {
    if (text.empty()) {
        throw std::invalid_argument("empty input");
    }
    return std::stoi(text);
}
```

Tests can explicitly verify failure paths by asserting that an exception is thrown.

5.4.2 Exception Boundaries

Exceptions should not leak across architectural boundaries unintentionally.

```
std::optional<int> safeParseInt(const std::string& text) {
    try {
        return parseInt(text);
    } catch (...) {
        return std::nullopt;
    }
}
```

This design isolates exceptions while preserving observability.

5.5 Error Codes and Testability

Traditional error codes can be testable if designed carefully.

5.5.1 Poorly Designed Error Codes

```
int parseInt(const std::string& text, int* result);
```

This design:

- Splits output across parameters
- Requires manual validation
- Encourages misuse

Tests must handle multiple failure modes implicitly.

5.5.2 Structured Error Codes

```
enum class ParseResult {
    Success,
    InvalidInput
};

ParseResult parseInt(const std::string& text, int& value);
```

This approach restores clarity and improves testability.

5.6 Expressive Error Types

Modern C++ encourages expressive error types that encode meaning.

5.6.1 Using Strong Types

```
struct ParseError {
    std::string message;
};

std::variant<int, ParseError> parseInt(const std::string& text);
```

Tests can now assert not only that an error occurred, but which error occurred.

Expressive error types improve diagnostics and test precision.

5.7 Avoiding Error Handling in Destructors

Errors in destructors are particularly harmful to testability.

```
class Resource {
```

```
public:
    ~Resource() {
        // throwing here is undefined behavior
    }
};
```

Such errors are difficult to observe and test.
Destructors should never fail. If failure is possible, it must be handled earlier.

5.8 Error Handling as Part of the API Contract

Error behavior is part of the public contract of a function.
When error handling is explicit, tests become simpler and more honest. When it is implicit, tests rely on assumptions.
A well-designed API makes success and failure equally visible.

5.9 Choosing an Error Strategy

There is no single correct error-handling strategy. The correct choice depends on context.
However, testable systems share common traits:

- Errors are explicit
- Failure paths are observable
- Contracts are unambiguous
- Behavior is verifiable

Error handling is not an implementation detail. It is an architectural decision.

When errors are designed for observability, tests become reliable. When errors are hidden, tests become guesses.

Testable C++ systems choose clarity over convenience.

Chapter 6

Designing Test Boundaries

Not everything needs to be tested equally. Attempting to test everything at the same level leads to slow, fragile test suites that provide little architectural confidence.
In long-lived C++ systems, the most valuable tests are those that verify behavior at meaningful boundaries. Boundaries separate what changes frequently from what changes rarely, what is deterministic from what is volatile, and what is core logic from what is infrastructure.
This chapter explains how to identify test boundaries, why boundaries matter more than internals, and how to design C++ code so that tests can exercise behavior without depending on real infrastructure.

6.1 What Is a Test Boundary?

A test boundary is a point in the system where responsibility changes.
Common boundaries include:

- Business logic vs. infrastructure
- Pure computation vs. I/O

- Application logic vs. operating system
- Deterministic code vs. nondeterministic services

Testing across boundaries increases cost and fragility. Testing at boundaries increases confidence.
Well-designed boundaries allow tests to replace volatile components while preserving real behavior in the core.

6.2 Why Internals Are Poor Test Targets

Tests that focus on internal implementation details are brittle.

6.2.1 Testing Internals: Fragile Design

```
class Calculator {
public:
    int compute() {
        intermediate_ = 10;
        return intermediate_ * 2;
    }

    int intermediate() const {
        return intermediate_;
    }

private:
    int intermediate_ = 0;
};
```

A test that asserts on `intermediate_` is coupled to implementation details. Any refactoring breaks the test without changing behavior.

Such tests discourage improvement and lock designs in place.

6.3 Testing at the Boundary

Boundary-focused tests assert observable behavior rather than internal state.

6.3.1 Boundary-Oriented Design

```
class Repository {
public:
    virtual ~Repository() = default;
    virtual int loadValue() const = 0;
};

class Application {
public:
    explicit Application(const Repository& repo)
        : repo_(repo) {}

    int run() const {
        return repo_.loadValue() * 2;
    }

private:
    const Repository& repo_;
};
```

The boundary here is clear:

- `Application` contains business logic
- `Repository` represents infrastructure

Tests can focus on application behavior without invoking databases, filesystems, or networks.

6.4 Replacing Infrastructure at Boundaries

Testable boundaries allow infrastructure to be replaced with simple substitutes.

6.4.1 Test Double for Infrastructure

```
class FakeRepository : public Repository {
public:
    explicit FakeRepository(int value)
        : value_(value) {}

    int loadValue() const override {
        return value_;
    }

private:
    int value_;
};
```

Tests can now verify behavior deterministically:

```
FakeRepository repo(21);
Application app(repo);
int result = app.run();  // result == 42
```

No database setup. No configuration. No cleanup.

6.5 Boundaries and Integration Tests

Not all tests should avoid infrastructure.
Integration tests verify that boundaries are wired correctly. However, integration tests should be fewer, slower, and more deliberate than unit-level tests.

6.5.1 Misplaced Integration Testing

```
Application app(realDatabaseRepository);
app.run();
```

Running this in every test introduces latency and instability.

6.5.2 Proper Role of Integration Tests

Integration tests should:

- Validate configuration
- Verify contracts between layers
- Run less frequently

Boundary-focused unit tests should carry most of the verification load.

6.6 Composition Roots

In testable systems, construction and wiring occur at composition roots.
A composition root is a place where:

- Concrete implementations are chosen
- Dependencies are assembled
- The application is started

6.6.1 Composition Root Example

```
int main() {
    FileRepository repo("data.db");
    Application app(repo);
    return app.run();
}
```

All construction is centralized. Business logic remains unaware of infrastructure.
Tests can bypass the composition root entirely.

6.7 Boundaries and Error Propagation

Boundaries are also the correct place to translate errors.

6.7.1 Error Translation at Boundaries

```
class DatabaseRepository : public Repository {
public:
    int loadValue() const override {
        if (!connected()) {
            throw std::runtime_error("database unavailable");
        }
        return 10;
    }
};
```

```
int Application::run() const {
    try {
        return repo_.loadValue() * 2;
    } catch (...) {
        return -1;
```

```
    }
}
```

Tests can now verify application-level behavior without depending on database failure modes.

6.8 Avoiding Boundary Leakage

Boundary leakage occurs when infrastructure concerns leak into business logic.

6.8.1 Leaking Infrastructure

```
class Application {
public:
    void run(DatabaseConnection& db);
};
```

This design couples application logic directly to infrastructure, making tests heavier and more fragile.
Boundaries must be enforced consistently.

6.9 Boundaries as Architectural Anchors

Boundaries serve as anchors for reasoning, testing, and refactoring.
They:

- Define what must be stable
- Isolate what may change
- Enable substitution without disruption

Well-designed boundaries allow systems to grow without collapsing under their own complexity.

6.10 Designing for Confidence

Test boundaries are not about reducing test count. They are about increasing confidence.
When tests exercise behavior at meaningful boundaries, engineers can refactor internals freely, knowing that externally visible behavior remains verified.
In testable C++ systems, boundaries are explicit, intentional, and respected.
Internals may change. Boundaries endure.

Conclusion

Testable C++ is not achieved through frameworks, libraries, or tooling. It is achieved through architectural clarity.

Throughout this booklet, one principle has remained constant: testability is not something that can be added at the end of development. It is a property that either emerges naturally from good design or is systematically prevented by poor design.

When dependencies are explicit rather than hidden, behavior becomes controllable. When state is localized and deterministic, behavior becomes predictable. When construction is intentional and centralized, behavior becomes observable and replaceable.

Under these conditions, tests stop being an obstacle. They stop being fragile, slow, or overly complex. They become straightforward expressions of intent.

C++ itself does not resist testing. On the contrary, C++ provides some of the strongest tools available for writing testable systems: deterministic object lifetime, explicit construction, precise ownership semantics, and zero-overhead abstraction.

What resists testing is ambiguity.

Hidden dependencies, global state, uncontrolled side effects, and implicit construction create systems that are difficult to reason about, difficult to test, and dangerous to change. Such systems often function correctly for long periods, but they do so without guarantees. Eventually, the cost of uncertainty exceeds the cost of development.

Well-designed C++ systems take a different path. They treat architecture as a first-class concern. They define clear boundaries. They isolate volatility. They make behavior observable

and failure explicit.

As a result, these systems can be verified continuously. They can be refactored safely. They can evolve without fear.

The goal of testable C++ is not to maximize test coverage or to follow testing trends. The goal is confidence.

Confidence that changes do not break existing behavior. Confidence that failures are detected early. Confidence that the system remains understandable long after its original authors have moved on.

That confidence is earned through design discipline.

C++ rewards engineers who think carefully about construction, ownership, and boundaries. When those principles are respected, testing becomes a natural and reliable companion rather than a constant struggle.

Testable C++ is not about writing more tests. It is about writing code that deserves to be tested.

And code that deserves to be tested is code that can endure.

Appendices

Appendix A: Testability Checklist

This checklist is intended to be used during design reviews, code reviews, and refactoring sessions. It does not replace architectural thinking, but it helps detect early warning signs that testability is being compromised.
Each item reflects a principle discussed throughout this booklet.

Dependency Management

- Are dependencies injected instead of constructed internally?
- Can each dependency be replaced in a test without modifying the production code?
- Are concrete implementations isolated from business logic?

Warning Signs

- Objects constructed with `new` deep inside functions
- Hard-coded file paths, URLs, or database connections
- Factories that always return concrete implementations

State and Lifetime

- Is state explicit and local to well-defined objects?
- Are object lifetimes deterministic and easy to reason about?
- Can tests create fresh instances without global resets?

Warning Signs

- Global variables or namespace-level mutable state
- Static objects with complex initialization
- Functions whose behavior depends on execution history

Observability

- Can behavior be observed through return values or explicit state?
- Are failure paths observable and testable?
- Do tests assert behavior rather than implementation details?

Warning Signs

- Functions that only produce side effects
- Silent failure or ambiguous return values
- Tests that inspect private internals

Volatile Concepts

- Are time, I/O, randomness, and external services abstracted?
- Can tests control nondeterministic inputs?
- Are volatile dependencies isolated at boundaries?

Warning Signs

- Direct calls to system time inside logic
- Random number generation without injection
- Business logic coupled directly to OS or network APIs

Boundaries and Architecture

- Are test boundaries clearly defined?
- Is infrastructure replaceable without affecting core logic?
- Are integration tests deliberate and limited?

Warning Signs

- Business logic directly manipulating infrastructure objects
- Tests requiring real databases or networks by default
- Widespread mocking of internal details

This checklist should not be treated as a rigid rule set. Its purpose is to prompt architectural discussion before problems become entrenched.

Appendix B: Common Anti-Patterns

This appendix documents recurring design mistakes that undermine testability in C++ systems. Each anti-pattern may appear convenient in isolation but causes disproportionate harm over time.

Global State

- Shared mutable state across the entire program
- Implicit coupling between unrelated components
- Order-dependent test failures

Global state introduces hidden dependencies and destroys isolation. Its presence is often justified by convenience and later regretted.

Hidden Singletons

- Implicit global access via static instances
- Inability to substitute implementations
- Tight coupling disguised as convenience

Hidden singletons are global state in disguise. They are particularly harmful because they appear controlled while remaining implicit.

Static Initialization Logic

- Unspecified initialization order across translation units

- Hard-to-debug startup failures
- Tests dependent on link order

Static initialization hides construction logic and removes control from both tests and application code.

Over-Mocking

- Mocking internal implementation details
- Tests that mirror production complexity
- Fragile tests that break during refactoring

Over-mocking is often a symptom of poor boundaries. Well-designed systems require fewer mocks and simpler test doubles.

Excessive Header-Only Dependencies

- Increased compile times
- Implicit coupling through templates
- Widespread rebuilds after small changes

Header-only designs are not inherently bad, but excessive use can entangle unrelated components and complicate testing and maintenance.

Treating Tests as Afterthoughts

- Designing APIs without considering testability
- Adding tests only after failures occur
- Viewing testing as a separate activity

Tests are not a secondary concern. They reflect the quality of the design that produced them. Avoiding these anti-patterns does not guarantee perfect design, but it dramatically increases the likelihood that a system remains testable, maintainable, and evolvable over time.
In professional C++ systems, architectural discipline is not optional. It is the foundation upon which testing, maintenance, and confidence are built.

References

This booklet is grounded in widely accepted professional practices, standards, and architectural principles developed over decades of experience in large-scale C++ systems. The references listed here are not intended as casual reading. They represent bodies of knowledge that informed the concepts, patterns, and examples presented throughout this work. No single document defines testable C++. Rather, it emerges at the intersection of language standards, toolchain behavior, architectural discipline, and industrial practice.

ISO C++ Standards and Guidelines

- ISO C++ Standard specifications (C++11 through C++23), with emphasis on object lifetime, RAII, error handling, and deterministic behavior
- ISO C++ Core Guidelines, focusing on ownership, resource management, interfaces, and architectural best practices
- Standard Library documentation for facilities related to error reporting, optional values, variants, and time abstractions

These documents define the language semantics that make testable C++ possible and establish the baseline for correct and portable design.

Large-Scale C++ Design and Architecture

- Case studies of long-lived C++ systems in finance, telecommunications, embedded platforms, and operating systems
- Architectural analyses of modularity, dependency management, and subsystem boundaries in native codebases
- Refactoring strategies for legacy C++ systems under continuous operation

These sources inform the architectural emphasis on boundaries, construction, and controlled dependencies.

Industrial Testing Practices for Native Systems

- Testing strategies used in safety-critical and performance-critical C++ environments
- Design-for-testability practices in systems programming and high-reliability software
- Approaches to balancing unit, integration, and system testing in native code

Industrial testing practice emphasizes predictability, isolation, and confidence over superficial coverage metrics.

Compiler, Toolchain, and ABI Documentation

- Compiler documentation for GCC, Clang, and MSVC, particularly regarding optimization, inlining, and debugging behavior
- ABI specifications affecting object layout, calling conventions, and exception handling

- Linker and build-system behavior impacting static initialization and test isolation

Understanding toolchain behavior is essential for designing code that behaves consistently across builds and environments.

Professional C++ Architecture Literature

- Books and papers on C++ architecture, resource management, and large-system design
- Publications on dependency management, modular design, and maintainability in native languages
- Technical essays and conference material discussing the evolution of C++ design practices

This literature provides historical context and reinforces the architectural principles that underpin testable C++ systems.

Closing Note

The intent of this references section is not to prescribe a fixed reading list, but to acknowledge the foundations upon which this booklet is built.
Testable C++ is not a trend. It is the result of disciplined engineering, informed by standards, experience, and hard-earned lessons from real systems.