# NO MORE HELLOWORLDS

# BUILD A REAL C++ APP

Architecture, Version control,
Testing and CMake
in One Practical Workflow

Nikolai Kutiavin

# No More Helloworlds
# Build a Real C++ App

Master Real C++ Development:
Architecture, Git, Testing & CMake
in One Practical Workflow

Nikolai Kutiavin

July 19, 2026

First edition.

For information, updates, and additional resources, visit:

https://sqglobe.com

# Contents

# 1. Introduction

C++ development is not only about the language itself. In real projects, it also includes software architecture, design principles, testing, build systems, and version control. These parts do not exist in isolation: they form a single development workflow. Architecture defines how a project is divided into decoupled modules. The build system and the language itself help enforce the boundaries between those modules. Tests and Continuous Integration (CI) then help ensure that changes violating those boundaries do not enter the main development branch.

Understanding this full workflow is one of the key differences between writing small programs and building software professionally.

When I started learning C++, I did not have an experienced developer to guide me. As a result, I made many avoidable mistakes and rewrote my pet projects far too many times. This book is written for readers who want to avoid that path and develop their skills more efficiently.

## 1.1 Why read this book?

If your goal is to understand how professional C++ applications are designed, built, and tested, this book is intended to guide you step by step. It is aimed at junior C++ developers who already know the basics of the language and want to learn how professional projects are structured.

Throughout the book, examples also use standard library types such as `std::string` and `std::ifstream`. If you need to review these components, https://en.cppreference.com is a useful reference.

This book brings together the main practices required to develop a professional C++ application. By following them, you will learn how to start new projects, structure code clearly, and avoid common design and implementation mistakes.

After reading this book, you should be able to:

- structure a modular project,
- add meaningful tests,
- isolate dependencies,
- configure CI,
- manage code changes with Git,
- enforce module boundaries with CMake.

The emphasis throughout the book is practical. Alongside theoretical explanations, you will find guidance based on real development experience. For example, significant attention is given to interface design: what should be considered when defining abstractions, how dependencies should be isolated, and how these decisions improve testability. A large part of the book is also devoted to testing, from unit tests to integration tests and manual validation. Taken together, these techniques help keep code reliable even after substantial refactoring.

Knowing the language is necessary, but professional development also requires an understanding of architecture, build systems, testing, and version control.

## 1.2 About the author

I am Nikolai Kutiavin, the author of this book. I have spent more than a decade building reliable, effective, and maintainable C++ software in several domains, including telecommunications, audio processing, automotive systems, and navigation.

Over the years, I have worked on high-quality, well-tested software ranging from small libraries and proofs of concept to major components of large systems. My work has included software architecture, build system configuration, design, C++ implementation, and writing unit and integration tests. I have also analyzed problematic C++ projects and helped identify and resolve critical technical issues.

During that time, I saw many projects become difficult to maintain, where even a small modification could lead to broken builds and defects in unexpected parts of the system. In most cases, this happened because architectural boundaries had become unclear and hidden dependencies had formed between modules. That is one of the main reasons why I wrote this book: to explain how architecture, language features, tests, and version control form a single development workflow.

## 1.3 Example application

Programming concepts are usually easier to understand through a concrete example. Throughout this book, we will work with a single project called **match-search**. The repository is available on GitHub: https://github.com/sqglobe/match-search.

The application is intentionally small so that architectural and testing decisions remain visible. The goal is not to build a feature-rich product, but to expose engineering techniques clearly.

Using one consistent example makes it easier to see how architectural ideas are translated into real C++ code and how tests are structured around them. Although selected code fragments appear throughout the book, exploring the full repository will provide the best overall picture.

The **match-search** application is similar in spirit to the well-known **grep** utility: it recursively searches a given folder for matches to a specified regular expression. Unlike **grep**, however, **match-search** provides a graphical user interface.

To build the project, you will need the following dependencies:

- CMake and either Ninja or Make,
- a C++ compiler with support for `std::expected` (for example, GCC 14 or another compiler with sufficient C++23 support),
- Qt6,
- GTest and GMock.

This book uses Linux-style paths and commands in many examples. The project can also be built on Windows and macOS, but the exact setup commands differ. If you want a reproducible environment without installing every dependency manually, use the development container described later in Section 2.7.4.

On Ubuntu 24.04 (Noble Numbat), the required dependencies can be installed with the package manager. The following command installs CMake, Ninja, and GCC 14:

```
sudo apt-get install -y cmake gcc-14 ninja-build
```

The next command installs Qt6 and the required development dependencies:

```
sudo apt-get install -y qt6-base-dev build-essential libgl1-mesa-
    dev
```

Finally, the following command installs GTest and GMock, which are required for the test targets:

```
sudo apt-get install -y libgtest-dev libgmock-dev
```

On other Linux distributions, package names and installation commands may differ.

**Tip**

On Windows, the project can be built with Visual Studio and MSVC if Qt6 and GTest/GMock are available to CMake. On macOS, Apple Clang or another modern compiler can be used, provided it supports the required C++ standard library features.

The **match-search** project is intentionally simple. It could, in principle, be implemented as a much smaller codebase, perhaps even as a single class. That simplicity is deliberate. It allows the book to focus on essential design ideas such as modularity, mocking, and dependency injection without adding unnecessary complexity. These concepts are easier to understand in a small project first and can then be transferred to larger and more demanding systems.

With the example project defined, the next step is to discuss its architecture: how to divide the application into modules, how to define boundaries between them, and how to keep the resulting codebase maintainable as it grows.

# Acknowledgements

I would like to thank Alexey Larikov for reading early drafts of this book and for providing thoughtful feedback on core topics such as architecture. His comments helped sharpen several explanations and make the material clearer and stronger.

I am also grateful to Ivan Dalidchik and Aniol Marín Atarés for their valuable feedback on the manuscript. Their comments helped me identify unclear parts, reduce ambiguity, and improve the overall structure of the book.

I would also like to thank Sasa Kostic, Matthew Isaacson, Shay Morag, Yoav Miller, Adrian Scillato and Michael Jørgensen for reading the early chapters and sharing their first impressions. Their feedback helped shape the book before the full draft was completed.

Any remaining mistakes are my own.

# Praise for the Book

The author's goal is to give readers the essential knowledge needed to build applications using modern programming techniques. The book guides readers through the full development cycle: from application design to building, testing, and version control.

This foundation is especially valuable for beginners. The book focuses on the overall development process and therefore complements resources that concentrate primarily on algorithms or C++ language features.

*– Ivan Dalidchik*
*Senior Software Developer at TrustMotion (former TTTech Auto)*

# 2. Architecture

Architecture is the skeleton of a project. Once an application is actively being developed, changing its architecture becomes difficult because structural changes usually trigger widespread rewrites. For that reason, it is worth designing the high-level architecture before writing the first lines of code.

This step gives you a clear overview of how the project will be organized: how the source code will be divided into modules, what each module is responsible for, and how those modules will interact. With that blueprint in place, development becomes smoother because the project already has a clear direction.

Investing time early to divide a project into logical components pays off quickly. Although it is possible to reorganize the structure later, doing so during active development is more complicated and more error-prone. It is much easier to prevent architectural chaos than to repair it after the fact, which is why architectural planning should come first.

As mentioned earlier, this book uses a simple GUI application called **match-search** as its running example. The project is available at `https://github.com/sqglobe/match-search`. In this chapter, I will use it to demonstrate a typical architectural design process.

The application allows the user to enter a regular expression and select a directory. It then recursively searches files in that directory, finds lines that match the given pattern, and displays the results to the user.

The project is intentionally small, which lets us focus on fundamental concepts of C++ application architecture without being distracted by complex business logic. In the following sections, we will explore the key architectural questions that shape the design:

- How should the project be divided into modules?
- How should the modules communicate with each other?

- How should errors be handled?
- How can the physical project structure reflect the architecture?

We will begin by discussing how to visualize the overall architecture so that design decisions become clear not only to you, but also to anyone collaborating on the project.

## 2.1 Role of the Architecture

Architecture defines the structural skeleton of a project: how responsibilities are divided, how components depend on each other, and where boundaries are drawn. With a sound architecture, the codebase remains stable, change is localized, and adding features stays predictable. With a weak architecture—unclear module responsibilities and blurry boundaries—every change tends to ripple through unrelated parts of the system, raising maintenance cost and increasing the risk of regressions.

A good architecture typically provides:

- **Clear responsibility split** — each part of the system owns a well-defined task and a clear public API.
- **Loose coupling between modules** — implementation changes in one module should not force refactoring in others.
- **Low maintenance cost** — code is easier to read, test, debug, and evolve over time.
- **Scalability** — new features can be added by extending the system, not by rewriting it.

This is especially important in C++. The language gives you a lot of power and flexibility, but without disciplined structure that flexibility often turns into fragile code, tangled dependencies, and costly refactoring.

In short, architecture is how you keep complexity under control as the project grows.

In the next sections, we will discuss how to split an application into modules based on responsibility, how to define architectural boundaries, and how to *enforce* them in practice. Before that, we introduce a tool commonly used to model and communicate architecture: **Unified Modeling Language (UML)**.

## 2.2 Diagrams

A clear and well-structured architecture makes an application far easier to develop and evolve. For that reason, it's important not only to design the architecture carefully but

also to document its essential aspects. The **Unified Modeling Language (UML)** is a powerful tool that greatly supports this process.

Once your architecture is defined, it must be communicated — to teammates, reviewers, and even your future self. UML diagrams make that possible. By visualizing structure and interactions, they transform abstract designs into something concrete and easy to reason about. Whether you're modeling components, workflows, or data flows, diagrams help make your decisions explicit and traceable. In this section, we'll explore the tools and techniques available to achieve that.

### 2.2.1 What is UML

As the saying goes, "a picture is worth a thousand words." **UML (Unified Modeling Language)** is a standardized modeling language for visualizing, describing, and documenting various aspects of software systems, business processes, and other structured models. It provides a consistent set of diagram types composed of visual elements such as arrows, shapes, and icons, which can be used to depict the most important parts of a system — for example, how different components of a complex system communicate with one another.

As a project grows, having a diagram that shows how classes relate to each other — where interfaces are defined, and which components depend on them — makes it much easier to understand and evolve the system. It simplifies later code analysis and modification, while relying solely on reading C++ files can quickly become overwhelming and error-prone.

While it's always possible to use custom icons or visual notations, these often require extra explanation. By contrast, the standardized set of UML elements ensures that your diagrams can be understood immediately by anyone familiar with the notation — without lengthy documentation.

To keep this understanding consistent, UML defines several types of diagrams, each describing a different aspect of a system:

- **Sequence diagram** — shows how and in what order objects communicate by calling methods or sending messages.
- **Class diagram** — depicts relationships between classes, interfaces, and structures.
- **Component diagram** — illustrates how the system is divided into structural components and how those components interact.
- **Activity diagram** — models the flow between activities, representing the order of operations.

- **State diagram** — describes the states of a system or object and the transitions between those states.

In practice, the most commonly used diagrams during software design are the **Sequence**, **Class**, and **Component** diagrams. These capture the core structural and behavioral aspects of most systems, so we'll focus on them in this section.

**Tip**

Use UML selectively — to emphasize architectural patterns, module relationships, and key interactions rather than every implementation detail.

There are available multiple instruments for the UML diagrams: Microsoft Visio, ArgoUML, Dia and may others. I personally use PlantUML (https://plantuml.com/). It introduces a simple language and generates diagrams out of it. The most diagrams in this book are created with that tool.

Keep in mind that UML is not meant to represent every detail of an application. For instance, a class diagram should not include every class in your codebase — that would quickly make it unreadable. Instead, focus on the key elements that define the system's architecture, such as module boundaries, inter-module communication, and dependencies between major components.

UML is not about drawing boxes — it's about clarifying thought. In the following sections, we'll see how each diagram type reveals a different dimension of your system's architecture.

### 2.2.2 Sequence Diagram

A **Sequence diagram** illustrates how and in which order objects communicate with each other by invoking methods or exchanging messages arranged in chronological order. It can represent interactions not only between individual objects but also between libraries, subsystems, or even separate programs. Sequence diagrams are particularly useful for illustrating how a system behaves in specific scenarios — for example, how it responds when an error occurs or when multiple events happen simultaneously.

Sequence diagrams are valuable for visualizing dynamic interactions between components. They clearly show which object initiates a process, how data flows between participants, and where dependencies or inefficiencies might arise. Before implementing a complex feature, creating a sequence diagram often reveals missing steps, redundant operations, or tight coupling that wouldn't be obvious from reading class definitions alone.

In the **match-search** application, Diagram 2.1 shows the interaction sequence when a user initiates a search. In this scenario, the user starts a search, receives notifications about found matches, and finally gets a completion notification when the process finishes.

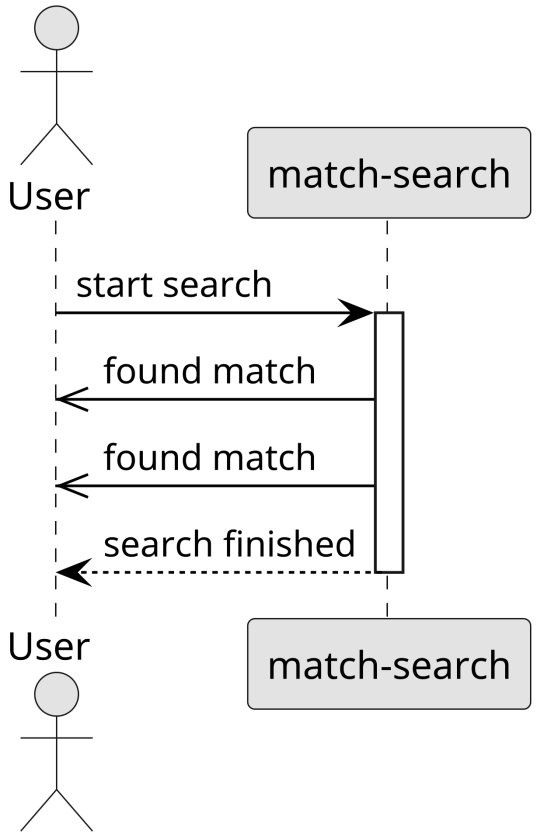



Diagram 2.1: Communication between user and **match-search**

Diagram 2.1 includes several standard UML elements:

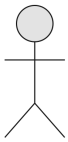
- **Actor**
- **Participant**
- **Activation box**
- **Request**

- **Response**

**Tip**

When designing a new feature, start with a sequence diagram — it helps you reason about interactions before defining the class structure.

An **Actor** represents an external entity that interacts with the system. In the case of **match-search**, the actor is the user, represented by the following symbol:

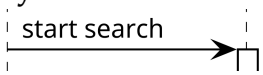
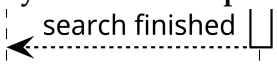
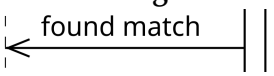


A **Participant**, in contrast, represents an internal part of the system. It can be a single object, a subsystem, a library, or even an entire application. A typical sequence diagram includes multiple participants, visualized as follows:


The **Activation box** represents the time during which an object or component is performing an action. In Diagram 2.1, it indicates the duration of an individual search request:

An activation box can also denote any form of lifetime, such as the existence of an object or a background operation.

Sequence diagrams use various types of arrows to indicate communication:

- **Synchronous message** — a function call where the sender waits for a response:
  
  
  Typically represents a regular C++ function invocation.
- **Synchronous response** — the return signal after the function completes:
  
  
  Usually corresponds to the function's return statement.
- **Asynchronous message** — a non-blocking call where the sender continues without waiting:
  
  
  This might represent an asynchronous event or background notification.

In Diagram 2.1, the **start search** request is synchronous — the user cannot trigger another request until it completes. Meanwhile, the **found match** notifications are asynchronous, allowing the application to continue working even if the user has not yet processed the update. Finally, a synchronous **search finished** response signals that the process has completed and the user can initiate another request.

As shown above, the term *message* is used in a broad sense — it simply means that information is transferred between entities. For instance, the user may not literally send a message to the system but may instead enter data through the UI. Sequence diagrams can therefore be applied to visualize any kind of data transfer, such as a function call, a TCP packet, or even manual user input.

**Tip**

Use sequence diagrams only for the most meaningful scenarios, and limit the number of actors, participants, and messages to what's essential.

Sequence diagrams are not meant to illustrate every possible flow. Use them when a scenario is complex or difficult to explain verbally — in such cases, a visual diagram clarifies interactions far better than text. However, keep them concise and up to date; overly detailed or outdated diagrams can quickly become a maintenance burden.

While a sequence diagram captures *how* components interact over time, it doesn't describe *what* those components are or how they relate structurally. That's the role of the **component diagram**, which we'll examine next.

### 2.2.3 Component Diagram

Before diving into implementation, it's essential to step back and understand how your system fits together at the module level. A **Component diagram** helps visualize this "big picture" — how subsystems interact, which parts depend on others, and where architectural boundaries should be drawn.

This high-level perspective prevents tight coupling, simplifies refactoring, and serves as a blueprint for long-term scalability. Like the **Class diagram**, the **Component diagram** belongs to the family of structural UML diagrams, but it operates at a higher level of abstraction. While a Class diagram focuses on relationships between individual classes and interfaces, a Component diagram captures the structure and dependencies among larger building blocks such as subsystems, modules, or libraries.

The real strength of a Component diagram lies in showing how the system's modules are organized and how data or control flows between them. Once these relationships are visible, it becomes easier to make sound architectural decisions and detect potential circular dependencies early on.

An example of a simple Component diagram is shown in Diagram 2.2.

In this example, three components interact with each other:

- The **Database** provides an interface named **SQL**.
- The **Server** consumes this interface to query data based on user requests and exposes an **HTTP** interface.
- The **Browser** sends requests through the **HTTP** interface.

The key elements of a Component diagram are:

- **Component** — A distinct part of the system that functions as an independent unit. In this example, the components are **Database**, **Server**, and **Browser**.
- **Interface** — Defines the functionality provided by a component and made available to others. Here, the interfaces are **SQL** and **HTTP**. Interfaces are not limited to network protocols — they can also represent internal APIs, C++ abstract classes, or any other defined contract between modules.

From this diagram, you can clearly see which interfaces are provided and consumed by each component, as well as the dependencies between them. This understanding is essential for maintaining, refactoring, or scaling the software while keeping module dependencies under control.

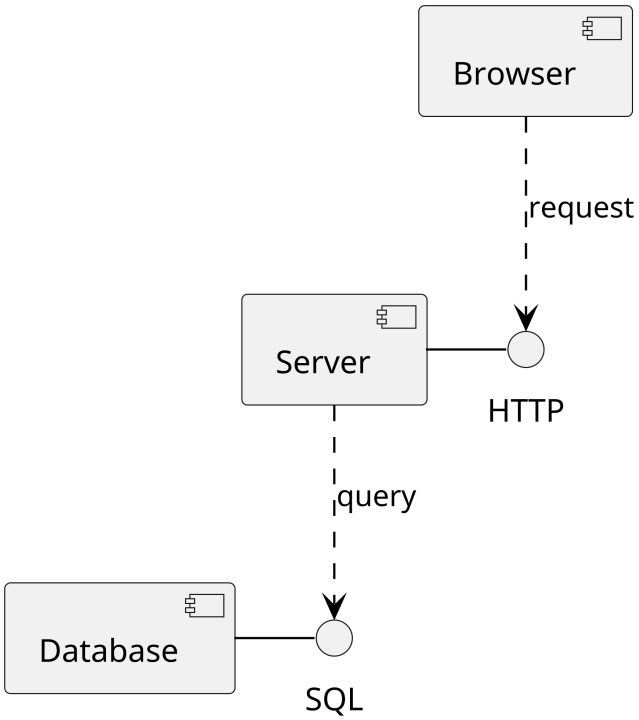



Diagram 2.2: Example of a Component diagram

**Tip**

Start by creating a **Component diagram** to visualize your system's main modules and their dependencies. Once the high-level structure is clear, move on to **Class diagrams** to describe each module's internal design.

In practice, I create a Component diagram for every major application I design. It helps me identify unwanted dependencies early — especially circular references — long before I write any CMake configuration or C++ code.

In C++ projects, a **Component diagram** can illustrate which modules expose particular interfaces and which modules consume them. This view makes it easier to analyze dependencies between libraries and uncover potential circular references at the design stage. Each component can later be expanded with a detailed Class diagram to show its internal composition.

Component diagrams effectively capture high-level relationships — how modules communicate and depend on one another — but they intentionally omit low-level implementation details. To explore the internal relationships within a module or library, refer to the **Class diagram** described in the previous section.

### 2.2.4 Class Diagram

Before implementing a design, it's critical to understand how different classes relate and depend on each other. A **Class diagram** provides that insight, mapping the system's static structure in a visual form. It represents classes and their relationships — including inheritance, composition, and dependencies — offering a clear, static view of how the system is organized. It may include classes, interfaces, exceptions, structs, and enumerations, along with their relationships such as inheritance, implementation, composition, aggregation, and dependency. In contrast to the **Sequence diagram**, which shows dynamic interactions, the **Class diagram** provides a static perspective of the system.

Class diagrams bring clarity and transparency at the library or module level — showing which classes are related, how they are composed, where public interfaces are defined, and which classes inherit from them. They form a solid foundation for implementation. By providing a high-level overview, class diagrams help uncover potential design issues before significant development effort is invested.

**Tip**

Create a class diagram before writing code to identify potential design issues early and validate your architectural choices.

Class diagrams are most effective when used as communication tools rather than strict blueprints. They allow teams to discuss design trade-offs — such as whether to use inheritance or composition — without committing to code prematurely.

As an example, consider the class diagram for a car shown in Diagram 2.3. The interface `ICar` defines the essential functionality of a car, while the class `RealCar` implements this interface. The `RealCar` class contains four `Wheel` and four `Door` objects. Additionally, the `Key` class is used to unlock the `ICar`, and each `Key` belongs to a `Person`.

The interface `ICar` defines several abstract methods that must be implemented by any derived class:

- `void startEngine()`

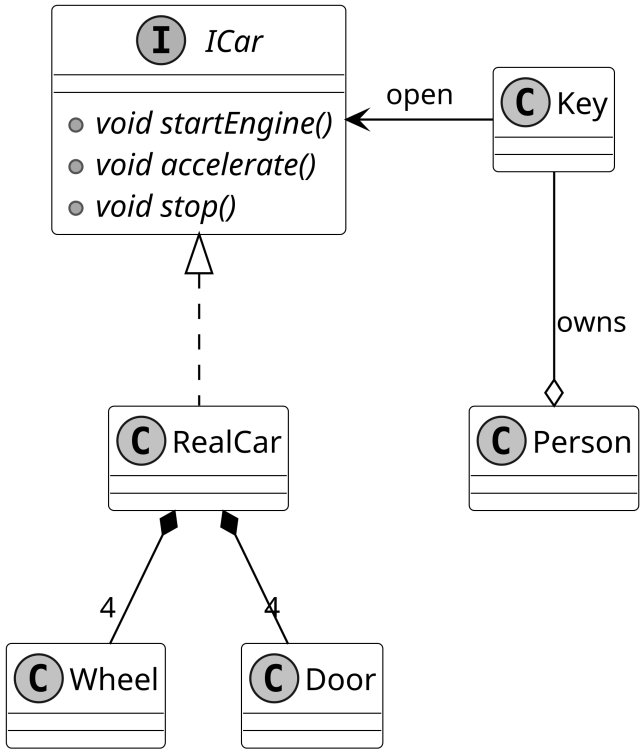



Diagram 2.3: Example class diagram for a car

- `void accelerate()`
- `void stop()`

Interfaces are marked with the symbol Ⓘ, while regular classes are marked with Ⓒ. Although it's possible to list all members and methods in a diagram, it's generally best to avoid doing so unless necessary — these details change frequently and can quickly make diagrams outdated and cluttered.

**Tip**

Focus on relationships between classes and interfaces rather than listing every field or method.

The class `RealCar` implements the interface `ICar`, indicated by an **Implementation** relationship. This is depicted with ┈┈▷, where the arrow points toward the implemented interface (`ICar`).

Furthermore, `RealCar` contains instances of the `Door` and `Wheel` classes. Since these objects cannot exist independently of the car, they are connected through a **Composition** relationship, shown with ◆———. The arrow points toward the container class (`RealCar`). You can also indicate multiplicity, such as one-to-one or one-to-many relationships. In Diagram 2.3, one `RealCar` includes four `Wheel` and four `Door` instances.

UML also defines a less strict relationship called **Aggregation**, which represents a "whole–part" connection where the parts can exist independently. In this example, the `Person` class has an aggregation relationship with the `Key` class. A `Key` can exist on its own, be copied, replaced, or shared among different people. Aggregation is represented by ◇———, with the arrow pointing to the aggregate (`Person`).

**Tip**

Use **Aggregation** when the "whole" and "part" can exist separately. Use **Composition** when the parts are tightly bound to the lifecycle of the whole.

Finally, the **Dependency** relationship describes situations where one object temporarily uses another, such as receiving it as a function argument. This is shown using ←———, with the arrow pointing to the object being used. In the diagram, the `Key` class depends on the `ICar` interface to unlock the car but does not store it as a member.

**Tip**

Avoid creating a single, large class diagram with every class and interface in the system. Instead, focus on key abstractions or create multiple smaller diagrams for each module.

While it may seem convenient to maintain one large class diagram for the entire system, such diagrams quickly become unreadable and difficult to maintain. A better

approach is to create smaller, module-specific diagrams that emphasize key interfaces and relationships.

Component diagrams define the architectural boundaries — showing what belongs where and how subsystems interact. Once those boundaries are clear, class diagrams describe the internal structure of each component. Together, they form a complete blueprint: from high-level architecture to detailed design. Sequence diagrams complement them by illustrating how components or classes collaborate over time. In later chapters, we'll use these diagrams to clarify different aspects of application development.

## 2.3 Splitting Project into Modules

Before diving into code, it's crucial to define how the project will be structured. Good modularization keeps the system understandable, testable, and adaptable as requirements evolve. UML provides the tools to visualize this structure, while architecture patterns offer guidance on how to organize it effectively.

Over the years, several architectural patterns have emerged to solve common design problems. Each has a different focus — some emphasize simplicity, others flexibility or modular independence. Among the most recognized are:

- **Layered Architecture** — separates the system into hierarchical layers, each responsible for a specific concern.
- **Hexagonal Architecture** — isolates the core logic from external dependencies using well-defined interfaces.
- **Microkernel Architecture** — centers on a minimal core with pluggable extensions for flexibility.

In this book, we will focus on the first two — Layered and Hexagonal — as they are the most relevant to C++ application development.

### 2.3.1 Layered Architecture

As the name suggests, **layered architecture** divides an application into distinct levels, each with a clearly defined responsibility. These layers form a stack: in a *closed* layered architecture, each layer may depend only on the layer directly below it and exposes its own interface to the layer above. This establishes a clear, one-directional dependency structure.

The key advantage of this approach is change containment. A modification within one layer usually affects only its direct consumers (typically the layer above), rather than propagating across the entire system. This localized impact makes refactoring safer and reduces the risk of accidental regressions.

A typical four-layer structure is illustrated in Diagram 2.4.

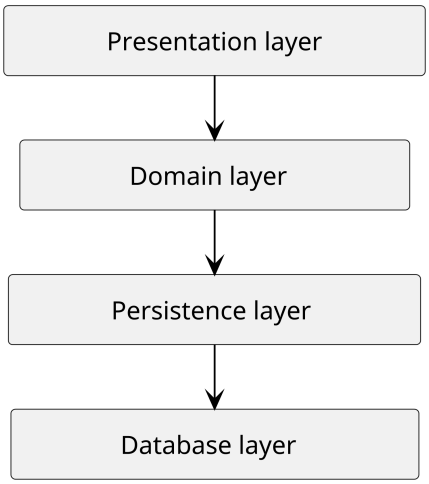



Diagram 2.4: Typical layered architecture

Each layer groups related functions and classes that implement meaningful behavior rather than merely passing data through. It exposes a *minimal* API—a small set of types and operations—that the layer above can use. For example:

- The **Presentation layer** handles user interaction: rendering the UI, processing input, and displaying results.
- The **Domain layer** implements core application logic: coordinating actions and enforcing rules.
- The **Persistence layer** manages data access: storing, retrieving, and caching information as needed.
- The **Database layer** provides physical storage, such as a relational or document-oriented database.

### How Layers Interact with Each Other

Consider a note-taking application. The **Presentation layer** provides the user interface: buttons and input fields to create, delete, and view notes. This layer knows how to display data and how to trigger the business logic. It has no direct dependency on the **Persistence layer** and does not access the database.

The **Domain layer** contains validation and coordination logic: a note must not be empty, tags must be unique, and the title must be present. It does not depend on the **Presentation layer**; it simply executes the requested operation. This layer communicates only with the **Persistence layer** and has no direct access to the database or UI.

Finally, the **Persistence layer** is responsible for storing notes: creating new records, updating existing ones, and deleting them. Only this layer talks to the database. Even then, **persistence layer** typically hides the database behind an abstract interface so it remains an implementation detail. The persistence layer does not know about the UI and must not call business-layer or presentation-layer code.

A typical execution flow looks like this (see Diagram 2.5):

1. The user triggers *Save* (for example, by clicking a button).
2. The **Presentation layer** calls a function in the **Business layer**, passing the input data.
3. The **Domain layer** validates the note and calls the **Persistence layer** to store it.
4. The **Persistence layer** translates the request into database operations and interacts with the **Database layer**.

As it was stated before, in a *closed* layered architecture, each layer depends only on the one directly beneath it. Each layer encapsulates its internal logic and exposes only a stable interface to the layer above. For example, the **Domain layer** may use a concrete type such as `NoteService` internally, but it should expose a clean API to the **Presentation layer** through an abstraction such as `NoteController`. Business-layer code should not manipulate UI widgets directly or show dialogs; if UI feedback is required, it should be done indirectly via an interface (see Section 4.2.2).

Likewise, the **Presentation layer** should call only the **Domain layer**, never the **Persistence layer** or the database directly. Similarly, the **Persistence layer** must not depend on the **Domain layer**. When these rules are violated, as shown in Diagram 2.6, the system becomes tightly coupled and fragile.

When boundaries are bypassed, implementation changes in one layer can trigger unplanned refactoring across the system and introduce subtle regressions. Enforcing

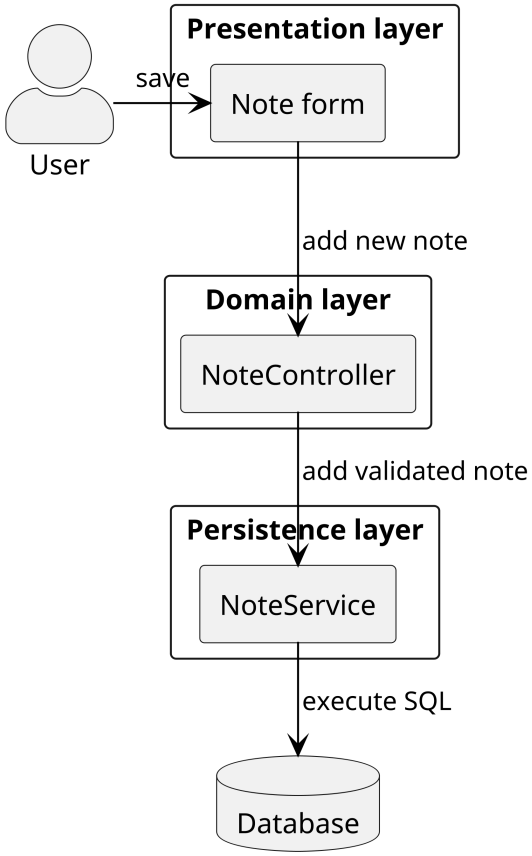



Diagram 2.5: Layer interaction flow for a note-taking application

layer boundaries prevents tight coupling and makes it easier to scale development across multiple contributors or teams.

**Important**

In a closed layered architecture, each layer should call only the functions and use the types of the layer directly below it. Breaking this rule undermines the integrity of the entire architecture.

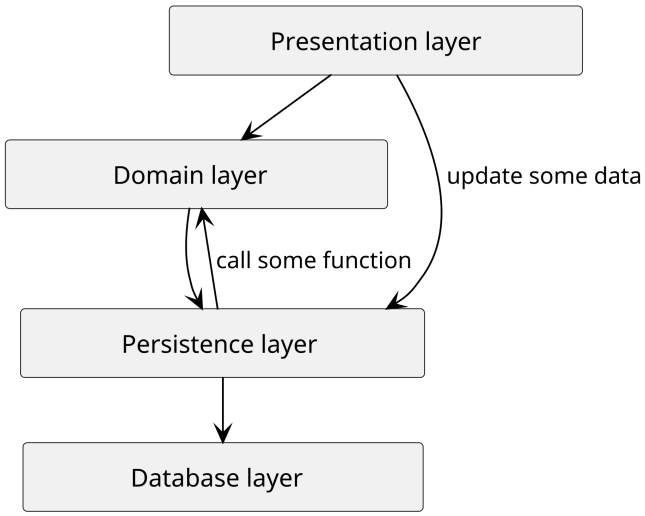



Diagram 2.6: Violation of layer encapsulation

**Why layers separation matters**

The separation when layer hides implementation and exposes only abstract interface also enables flexibility. If the SQL database should be replaced with document oriented one, the single place to be rewritten is only implementation of **Persistence layer**. If the interface `NoteService` remains the same business logic isn't even affected by the refactoring because it doesn't depend on the database directly. This modularity is illustrated in Diagram 2.7. Beyond modularity, this structure also enables easier testing and controlled evolution.

Another advantage of this isolation is that lower layers can evolve independently without impacting the rest of the system. For instance, the **Persistence layer** might introduce caching for expensive queries without affecting the **Domain layer**, which treats it as a black box. As long as the API remains stable, upper layers remain unaffected.

A key principle of this architecture is minimal exposure: each layer should reveal only what is strictly necessary — ideally through abstract interfaces rather than concrete implementations. This approach simplifies among other thins also the testing. During testing, for example, **Presentation layer** can be replaced with test code that directly invokes functions in the **Domain layer**. Similarly, the interface `NoteService` from **Persistence layer** can be replaced by a **Mocked class** that implements this interface but returns predefined data instead of querying a real database, as shown in Diagram 2.8.

This setup allows the implementation of **Domain layer** to be tested independently

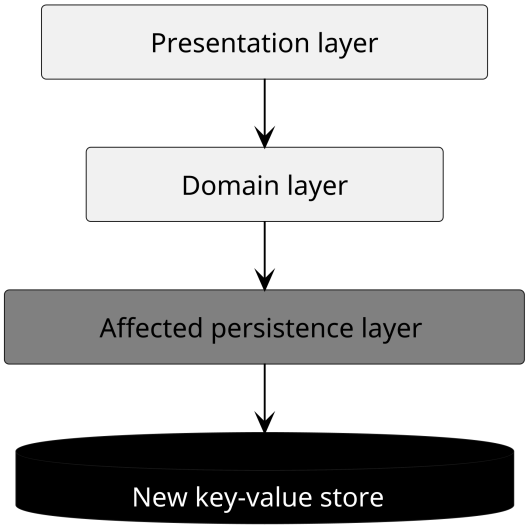



Diagram 2.7: Updating the Persistence layer affects only the layer above

of the database, making it easier to verify both normal and error-handling scenarios.

**Anti-pattern**

Even well-designed layered systems can degrade over time when boundaries are treated as formalities rather than contracts. One common failure mode is the *sinkhole anti-pattern*. In a layered design, the **Presentation layer** must not call the **Persistence layer** directly; requests should flow through the **Domain layer**. However, if the **Domain layer** adds no meaningful logic and merely forwards calls, it becomes a "sinkhole": an extra hop that increases complexity without adding value.

For example, when the user interface requests a note, an implementation of `NoteController` might simply forward the request to `NoteService`, then map the result into a presentation-friendly format—without performing any validation, coordination, or business rule enforcement. In that case, the layer exists mostly to satisfy a diagram, not a responsibility.

A practical way to detect this problem is the 80/20 heuristic: if roughly 80% of calls through the **Domain layer** involve real business logic (validation, orchestration, invariants, cross-cutting rules) and only 20% are simple pass-throughs, the layering is likely justified. If the ratio is reversed, the design may be suffering from the sinkhole

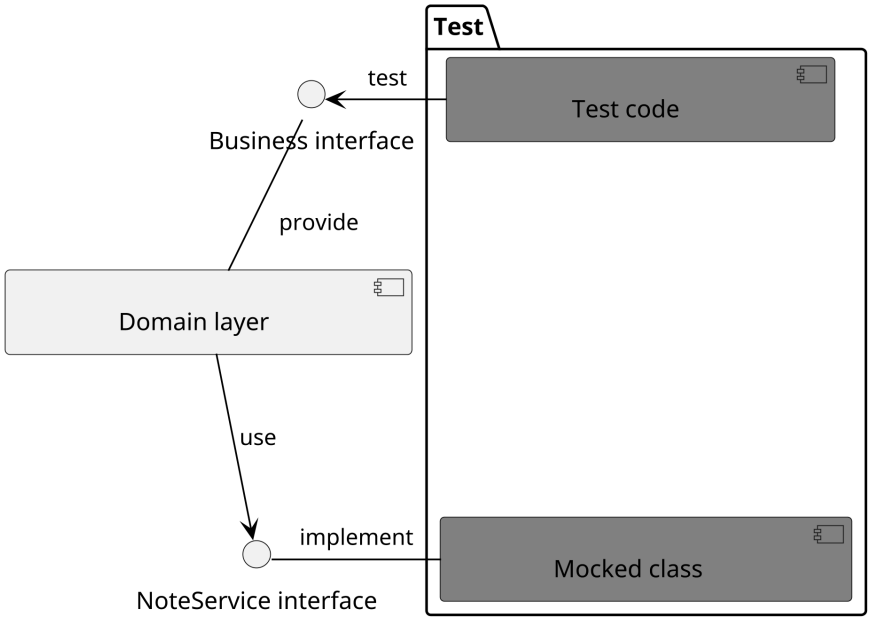



Diagram 2.8: Lower layers can be mocked for testing

anti-pattern, and the layer boundaries should be reconsidered.

**12 Rules for Layered Architecture**

When defining a layered architecture, consider the following rules:

1. Each layer must have one clear responsibility.
2. Dependencies must point downward only.
3. A layer may depend only on the public API of the layer below.
4. Cross-layer communication should use abstractions.
5. The executable is a composition root, not a logic layer.
6. Boundaries must be enforced physically: in folders, headers, and targets.
7. Public APIs should be minimal.
8. Implementation details must not leak across layer boundaries.

9. The testing strategy should follow the architecture.
10. Shared test helpers belong in a dedicated testing module.
11. A layer must add value; otherwise, it becomes a sinkhole.
12. Architectural rules should be enforceable by the build system and tests.

When introducing a new layer, ask these questions before writing the code:

- What responsibility belongs only here?
- Which layer is allowed to call it?
- Which lower-layer API does it need?
- What should remain hidden?
- How will this layer be unit-tested?
- Which dependency rule in CMake enforces the boundary?

The answers will help you understand the layer's boundaries more clearly and define them properly.

Layered architecture is often associated with web applications, but the same principles translate well to modern C++: clear modularity, testability, and separation of concerns. The next section explores how to adapt these ideas to C++ project structure and enforce boundaries in practice.

### 2.3.2 Applying Layered Architecture to the match-search Project

After exploring the principles of layered architecture, let's see how they translate into real-world C++ code using the **match-search** project as a practical example. At first glance, layering might seem unnecessary here, especially since there's no database involved. However, the principles of layered design remain just as valuable for small utilities as for large applications.

For the **match-search** project, the application can be logically divided into the following layers, which separate responsibilities, reducing coupling and making the system easier to test and evolve:

1. **Presentation layer** — the graphical user interface (GUI) that handles user interaction.

2. **Domain layer** — the core logic that performs file content searches using regular expressions.

3. **Filesystem layer** — the interaction with the underlying file system.

Together, these layers form a clear top-down flow: the user interacts with the GUI, which invokes business logic, which in turn accesses the filesystem.

In this setup, the **Filesystem layer** is implemented using standard C++ library components such as `std::filesystem` and STL I/O utilities. For a small project like **match-search**, abstracting the file system would be excessive — the file-handling logic is simple enough to work directly with the standard library. However, in larger systems (for example, a logging framework with log rotation or archiving), file operations often become more complex. In those cases, introducing a dedicated abstraction layer for filesystem access becomes essential to preserve testability and maintain a clean architecture. Without such abstraction, achieving high test coverage can be difficult due to the many edge cases involved.

In C++ projects, each layer is typically encapsulated as a separate library. Each library exposes a clear public API to the layer above while hiding its internal details and implementation logic. Following this principle, the **match-search** project is organized into the following libraries:

1. **gui** — corresponds to the **Presentation layer**.

2. **files-search** — represents the **Domain layer**.

3. **STL** — provides direct access to the file system and forms the **Filesystem layer**.

#### Prevent Leaking low-level Primitives

A crucial rule in this design is to prevent low-level primitives from leaking into higher layers. Such leakage breaks the abstraction boundary, forcing upper layers to depend on low-level details — exactly what layered architecture aims to prevent. For example, classes such as `std::filesystem::path` or `std::fstream` from the **Filesystem layer** should never appear in the public API of the **files-search** library. Otherwise, these types would "leak" into the **gui** layer, violating the principles of layered architecture.

For instance, consider the following incorrect implementation of the function exported by **files-search**, which trigger the overall search process. It takes a path to the root directory to start the search process and the regular expression to compare:

```
std::string search(
    const std::filesystem::path &whereToSearch,
```

```
    const std::regex &regularExpression);
```

In this case, the **gui** module would need to construct an `std::filesystem::path` object to call `search()`, directly depending on the **Filesystem layer**. This violation is illustrated in Diagram 2.9.

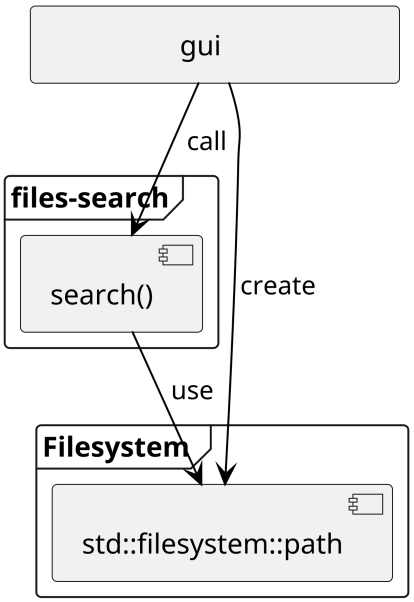



Diagram 2.9: The **files-search** module incorrectly exposes `std::filesystem::path`

To fix this issue, the function can instead use `std::string` parameters (Much more efficient implementation is discussed here Section 4.2.4):

```
std::string search(
    const std::string &whereToSearch,
    const std::string &regularExpression);
```

Now the **gui** module no longer depends on `std::filesystem::path` and is fully decoupled from the **Filesystem layer**. Internally, the **files-search** library can still convert the string into a `std::filesystem::path` when needed. This small change enforces proper separation between the business logic and filesystem access.

**Important**

Do not expose in a layer's public API any class or type defined in the layer below.

This separation of responsibilities is illustrated in Diagram 2.10.

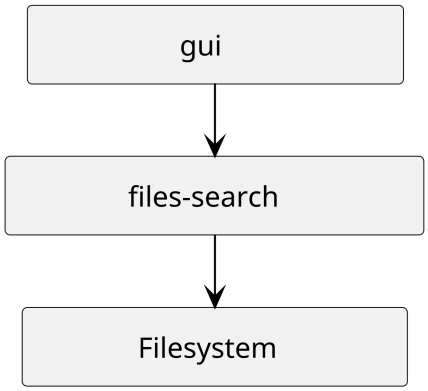



Diagram 2.10: Layered architecture in the `match-search` application

In this design, the **gui** layer should never directly read or write files. All file operations are handled by the **files-search** layer.

### Hiding Implementation by the Interface

Abstract interfaces are the best mechanism to expose functionality from one layer to another while keeping implementation details hidden. They formalize contracts between layers, allowing components to evolve independently without breaking each other and act as a bridge between the public API and internal logic. Such an interface declares which methods are available, without revealing how they are implemented. In C++, interfaces are discussed in detail in Section 4.1. For example, the **files-search** layer might expose a simple interface `Searcher` with a single method `search`, as shown below (Diagram 2.11):

```
class Searcher {
public:
    virtual std::string search(
        const std::string &whereToSearch,
        const std::string &regularExpression) = 0;
```

```
};
```

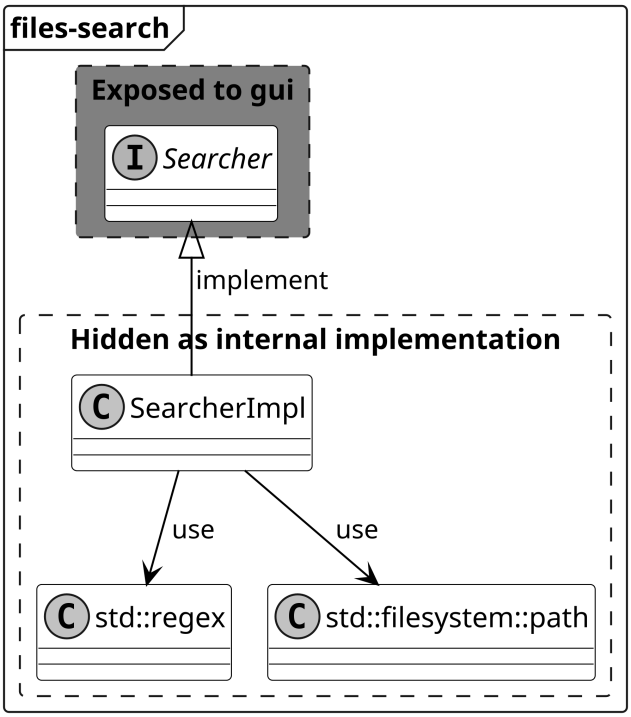



Diagram 2.11: Implementation is hidden behind interface

This method performs a recursive search for matches to the given regular expression within files under the specified path. It provides high-level functionality without exposing details about the regular expression engine or filesystem operations. All classes in the **gui** layer can depend solely on this interface — they don't need to know how the search is performed, which regex engine is used, or how file traversal is implemented. This separation formalizes dependencies and ensures that upper layers never rely on internal implementation details. Section 4.4 explains how to enforce those boundaries on the buildsystem level and in C++ code.

The main benefit of this scheme is flexibility. The implementation can freely change — switching between breadth-first or depth-first traversal, adding caching, or optimizing I/O — without affecting the **gui** code. As long as the interface remains the same, the system stays stable. Interfaces are a major topic and are discussed further in Section 4.2.

**Tip**

Expose only abstract interfaces from each layer to hide implementation details and facilitate testing.

### Testing for the match-search

Beyond modularity, interfaces also unlock one of the biggest advantages of layered design: simplified testing. In unit tests, the real implementation can be replaced by a dummy or mock class that returns predefined results instead of performing actual filesystem operations. This makes it possible to simulate hard-to-reproduce scenarios — such as corrupted files, broken links, or very large directories — and verify that the user interface handles these cases correctly. The approach is discussed in more detail in Section 5.4.

Conversely, testing the **files-search** layer itself is more challenging (see Section 5.5.2 as example), as it depends directly on `std::filesystem`, which lacks interface-based abstraction. Such tests must manipulate the actual filesystem, which adds some complexity but remains manageable for small tools like **match-search**. With this structure, both development and testing remain efficient, even as the project grows.

In conclusion, I chose the layered architecture for **match-search** because it is simple, clean, and fits the project's goals perfectly. With these boundaries in place, **match-search** demonstrates how layered design principles can be applied effectively even in small-scale C++ projects.

**Tip**

Layered design enforces clear boundaries between responsibilities, keeping systems modular, testable, and adaptable — even for small C++ projects like **match-search**.

That said, no single architecture is universally optimal. Always consider your project's size, requirements, and constraints before deciding on an approach. The next chapter introduces another architectural pattern that may better suit certain scenarios.

**Continue Reading**

The full book continues with hexagonal architecture, module communication, error handling, CMake, testing, Git, CI, Docker, Qt, and the complete implementation of the example application.

Learn more and get release updates:
https://sqglobe.com/no-more-helloworlds-build-a-real-c-app/