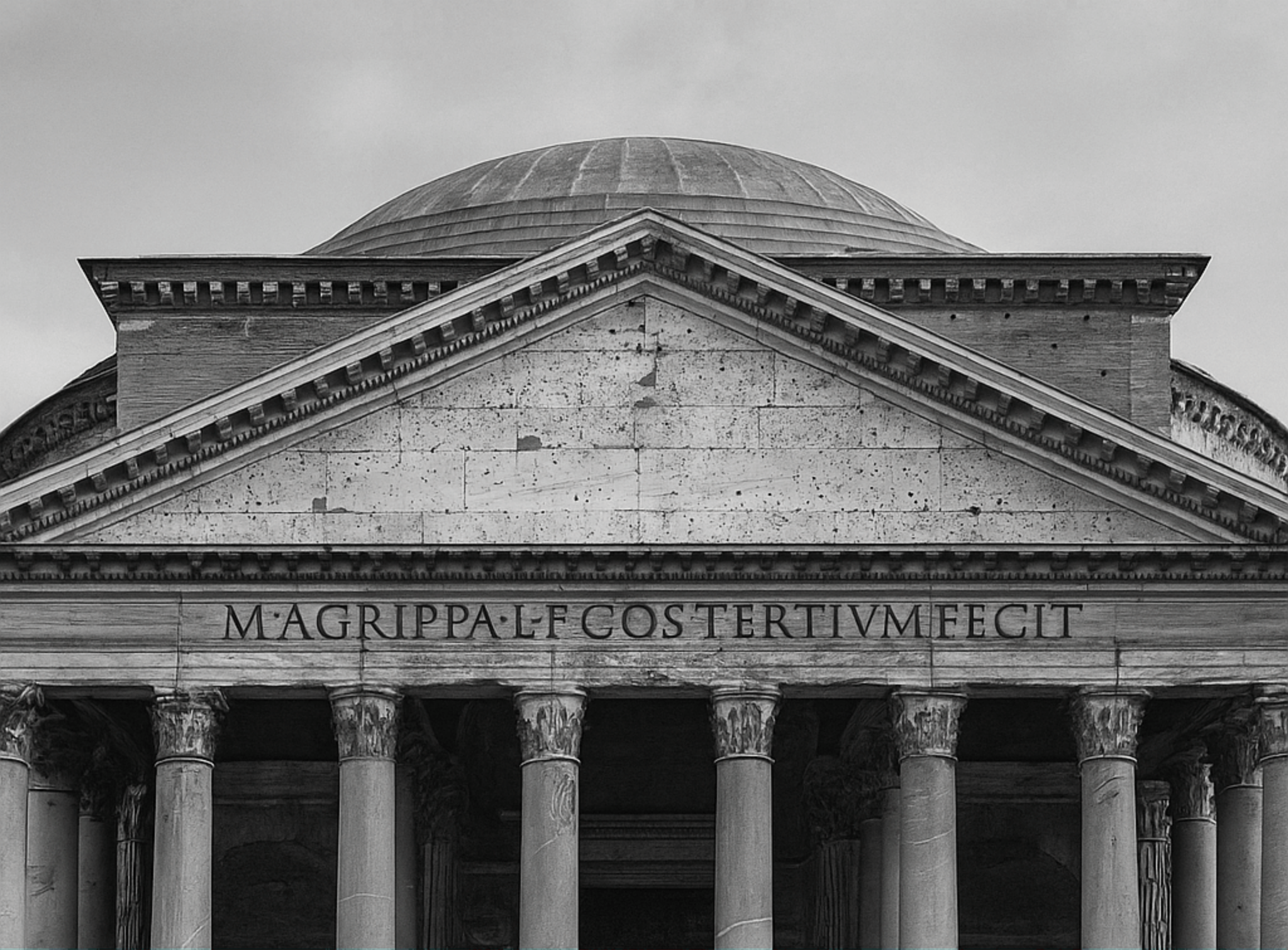

SAFE BY DESIGN

Explorations in Software Architecture and Expressiveness

Mykola Haliullin

TYPE-SAFE BY DESIGN

Explorations in Software Architecture and Expressiveness

Written by Mykola Haliullin

Introduction

This book is the result of an ongoing reflection on what it means to build software properly. From abstract thoughts on architecture to practical engineering patterns, it is an attempt to answer a deceptively simple question: How can we write code that remains correct, adaptable, and enjoyable to work with over time?

More than anything, this book is about how to think about the software development process. It's not just about what patterns to use, but when and why. Good software design isn’t about chasing purity or elegance—it’s about shaping systems that resist decay, support change, and make correctness easier than error.

There is a certain kind of joy in working on a well-structured codebase. When the architecture is sound and the guardrails are in place, each next line of code feels obvious. The compiler becomes your partner, catching mistakes before they reach production. The flow becomes so natural that, hypothetically, you could have a show running in the background while coding—and still avoid bugs.

This book proposes two core ideas:

- High-level structure: so that you always know where to put things, and why they go there.
- Low-level safeguards: so that modules are written correctly, and dependencies behave in predictable, verifiable ways.

When a change affects multiple components, the system should make this visible—not through documentation, but through failing to compile when something is missing. It should never be possible to break correctness silently, only to discover the consequences days later in production or, worse, on a customer's screen.

The chapters in this book are independent essays. You can read them in any order. But they are united by a common mindset: the belief that architecture is not about adding structure—it's about managing change. Not through heroics or process, but through thoughtful design, rigorous interfaces, and the subtle power of type systems.

This book is not a manual, and it is not a catalog of patterns. It is a curated set of essays—each one an exploration of a single problem, perspective, or constraint that affects how we build robust software. Some chapters dive deep into type systems and contracts. Others investigate design flexibility, change complexity, or structural clarity from less obvious angles. But all of them share one goal: to illuminate the invisible scaffolding that makes code correct by design, not by accident.

Some essays may seem unusual or domain-specific at first glance—a discussion of game abilities, or a strategy for constructor enforcement—but they all belong here. Each one reveals a different facet of the same principle: that software design is not about writing code that works once, but about building systems that keep working as they grow, change, and interact with other parts.

If there's a unifying theme, it’s this: type-safe design is not just about types. It’s about using language features, abstractions, and architectural patterns as tools to guide behavior, encode intent, and prevent failure at scale.

Applying Big O Notation To Software Design

As software systems grow in size and complexity, the cost of making changes can scale unpredictably. While we often rely on intuition and experience to judge design quality, this article proposes a more formal approach: applying Big O notation to software architecture.

By analyzing how the effort required to implement changes grows with system size, I introduce a new lens for evaluating rigidity, fragility, immobility, and other key architectural traits. This framework bridges traditional design principles—like SOLID and design patterns—with measurable structural behavior.

As software systems grow, so does the cost of modifying them. In many cases, a seemingly minor change leads to cascading updates across unrelated parts of the system. Despite being commonly labeled as “rigid,” “brittle,” or “difficult to change,” these terms remain vague without quantitative backing. By reinterpreting Big O notation in the context of software design, we can measure and communicate the true cost of change in a more precise and actionable way.

This perspective helps us distinguish between code that is inherently dangerous to evolve, and code that remains adaptable over time. Ultimately, the goal is to design systems in which the cost of making changes grows at a controlled rate, not

exponentially.

Although Big O notation is traditionally used to analyze algorithmic complexity, this chapter reimagines it as a tool for evaluating architectural cost. The connection to Type-Safe by Design lies in the shared goal: controlling the long-term behavior of systems. Just as type systems prevent invalid states, structural reasoning with Big O prevents systems from becoming unmanageable as they grow. This chapter builds the case for design as a mathematical discipline — not just an art.

Defining O(f(n)) for Code Modifications

In this article, I reinterpret Big O notation not in terms of runtime performance, but as a way to describe the **effort required to modify a codebase**.

Specifically, I define that a change in a program requires ***O(f(n))*** **effort**, where ***f(n)*** is the number of **logically isolated, necessary modifications** a developer must perform to implement that change. The variable ***n*** refers to the size of the system—not in terms of memory or lines of code, but in structural units such as components, modules, or classes.

Two implementations that deliver the same functionality may still differ in architectural quality if one requires significantly more isolated edits to implement a change than the other. I am particularly interested in the **asymptotic behavior** of this effort as the system grows.

In well-designed systems, important changes should ideally

require **constant or sub-linear effort**—O(1), O(log n), etc. When the number of required edits grows linearly or worse—O(n), O(n²)—this signals architectural fragility and poor scalability in terms of maintainability.

Rigidity: When Change Propagates Too Far

One of the most tangible manifestations of poor software architecture is **rigidity**—the tendency of a system to resist change. In practical terms, rigidity means that a single, localized requirement leads to changes across many unrelated parts of the codebase.

To evaluate this, I analyze the **growth rate of required modifications** relative to system size. If a codebase demonstrates O(1) or O(log n) rigidity, it means most changes remain localized, regardless of how large the system grows. This is a key property of scalable design.

However, when a system exhibits **O(n)** rigidity, a change in one part often forces changes across the entire system. Even worse, in systems with **O(n²)** rigidity, each new component increases the number of potentially affected places exponentially. In such cases, maintenance becomes a source of constant risk, delays, and bugs.

This concept aligns closely with the *Open/Closed Principle*: systems should be open to extension but closed to modification. In architectures with high rigidity, developers often find themselves modifying existing code instead of extending it—an anti-pattern that accumulates technical debt over time.

By modeling rigidity with Big O notation, I provide a language to discuss maintainability in formal terms. This allows teams to identify when architectural degradation is not just a matter of "feeling," but of **measurable structural behavior**.

Fragility: When a Change Breaks What Shouldn't Be Touched

Another common sign of a weak design is **fragility**—the tendency of seemingly safe changes to introduce unexpected breakages in unrelated parts of the system.

Fragility often appears alongside rigidity. While rigidity forces changes to propagate broadly, fragility means those changes are **dangerous**, even when made carefully. A fragile system behaves like a house of cards: touching one piece can bring down the rest.

I define fragility as the **likelihood that a logically isolated edit causes side effects in unrelated modules**. As the system grows, this risk should ideally remain bounded. If the probability or frequency of breakage increases with system size—approaching O(n) or worse—we're dealing with an architecture that **scales poorly in reliability**.

This concept complements Robert C. Martin's original definition of fragility as "the tendency of the software to break in many places when a single change is made." My contribution here is to interpret fragility not as an emotional developer experience, but as an **emergent structural property**, expressible through asymptotic

behavior.

In modern software engineering, especially in systems with long lifespans or high availability requirements, **minimizing architectural fragility is essential** to achieving sustainable development velocity.

Immobility: When Useful Code Can't Be Reused

Immobility refers to a system's inability to reuse existing components without incurring excessive cost. In rigid and fragile systems, useful code often becomes "trapped" inside tightly coupled modules, making it difficult to extract without introducing risk.

This concept, originally described by Robert C. Martin, becomes especially important in large codebases, where reusing existing functionality is essential for development speed and consistency. If a component is logically reusable but its actual extraction and integration elsewhere require modifying many files or understanding multiple layers of dependencies, then the cost of reuse grows with system size—sometimes **O(n)** or worse.

In high-immobility systems, engineers often resort to **duplicating logic** instead of reusing it. While this avoids immediate coupling, it introduces a long-term liability: duplicated behavior that must be maintained in multiple places. This leads to architectural drift, inconsistency, and eventual loss of control over the codebase.

By treating immobility as a **growth function**—the number of unrelated modules affected when trying to reuse a unit—we can

formally evaluate how reusable the architecture actually is. Reusability is not just a matter of intention; it's a matter of **change complexity**.

A scalable architecture enables reuse with constant or low-complexity operations. Anything beyond that becomes an obstacle to long-term maintainability and productivity.

Viscosity: When the Easier Path Is the Wrong One

Viscosity describes the degree to which a system encourages or discourages clean, principled changes. In a high-viscosity codebase, making the correct architectural change requires significantly more effort than applying a quick workaround or hack.

This concept, as introduced by Robert C. Martin, draws attention to developer behavior: when the **path of least resistance leads to poor design**, the system naturally degrades over time. In other words, if the architecturally correct change is **O(n)** in cost while the shortcut is **O(1)**, most developers—under time pressure—will default to the shortcut.

In such environments, even well-intentioned engineers may unintentionally contribute to architectural erosion. Clean layering, modular separation, and adherence to design principles become difficult to maintain when the structure actively resists principled modifications.

I propose thinking of viscosity as a **delta between the cost of a correct change and an incorrect but easier one**. When that

delta widens, architecture tends to degrade. To scale well, a codebase must not only support clean changes but must also **make them the path of least resistance**.

High-viscosity systems often accumulate a surface-level appearance of functionality, while underlying complexity and inconsistency quietly grow. Left unchecked, this results in systems that become prohibitively expensive to evolve or refactor.

A Case Study in Structural Decay: The Spread of Copy-Paste Logic

To illustrate the cumulative effect of rigidity, fragility, immobility, and viscosity, consider a real-world example I encountered early in my career.

The team needed to make a minor behavioral adjustment to how a certain button behaved in the UI. However, the original implementation had been copied across dozens of components with slight variations. No abstraction had ever been introduced. Each copy handled edge cases differently, and the underlying business logic was embedded directly within UI code.

At that point, what should have been a one-line fix became a **multi-day task**. Each instance had to be manually located, analyzed for differences, tested, and modified. The deeper we went, the more inconsistencies we uncovered—many of which had no clear ownership or rationale. The original design had no mechanism for extension, and the cost of centralizing behavior had grown prohibitively high.

This scenario is not unique. It's the natural outcome of an architecture with:

- *O(n)* rigidity—changes ripple through the entire system,
- *O(n)* fragility —each change carries the risk of introducing bugs,
- *O(n)* immobility—useful behavior is trapped inside monolithic structures,
- and high viscosity—developers chose copy-paste as the "cheapest" solution.

By the time these patterns are visible, reversing them requires significant investment. Recognizing them early—and being able to quantify them—is key to sustainable architecture.

Applying Big O to SOLID: SRP and ISP as Tools for Controlling Change Complexity

The **Single Responsibility Principle (SRP)** states that a module should have only one reason to change. This is often interpreted philosophically, but I propose viewing SRP as a mechanism for **controlling the complexity of change.**

When a class violates SRP, a single modification request may require edits across multiple unrelated concerns. This often results in ***O(n)*** or worse modification effort, where ***n*** is the number of responsibilities entangled within the module. By enforcing SRP, we reduce the number of unrelated edits and push the system closer to ***O(1)*** behavior for isolated changes.

Similarly, the **Interface Segregation Principle (ISP)** reduces the impact radius of change. If a client depends on a large, bloated interface, even a minor change to that interface can force downstream consumers to update —a classic case of **high fragility and rigidity**.

By breaking down interfaces into smaller, more focused units, ISP ensures that changes remain **localized**. In Big O terms, it shifts the cost of change from *O(n)*—where all clients must adapt—to ***O(1)*** or ***O(k)***, where k is the number of directly affected consumers.

These principles, often treated as stylistic recommendations, gain renewed strength when expressed through **asymptotic reasoning**. They are not just about elegance—they are about **making change scale sustainably**.

Extending Big O Reasoning to OCP, DIP, and LSP

The **Open/Closed Principle (OCP)** asserts that software entities should be open for extension but closed for modification. In practical terms, this means that new behavior should be added without altering existing code.

From a complexity standpoint, good adherence to OCP allows feature additions to remain ***O(1)*** —new modules can be introduced without modifying or retesting the old ones. Poor adherence, by contrast, leads to ***O(n)*** or worse, where every extension forces changes across many existing components. This undermines system stability and increases the cost of innovation.

The **Dependency Inversion Principle (DIP)** plays a complementary role. Without it, high-level modules often depend directly on low-level details. This creates **tight coupling**, leading to ripple effects when underlying implementations change.

By inverting dependencies and introducing abstractions, DIP **contains the blast radius of change**. Instead of updating both the core logic and implementation when requirements shift, we isolate changes to implementation-level classes—again pushing the system toward **constant time evolution**.

Finally, the **Liskov Substitution Principle (LSP)** ensures that subtype polymorphism does not introduce semantic surprises. Violations of LSP often result in **fragile polymorphic hierarchies**, where changes in base

classes break subclasses or vice versa. This forces developers to make defensive edits across multiple layers of inheritance—a behavior that grows worse with system size.

Viewed through the Big O lens, LSP violations convert what should be **localized behavior extensions** into **global refactoring problems**. This effectively transforms what should be ***O(1)*** modifications into ***O(n)*** cascades.

Taken together, OCP, DIP, and LSP are not just philosophical ideals—they are **concrete tools** for minimizing the complexity and risk of structural change. When properly applied, they preserve architecture's ability to scale, adapt, and evolve.

Design Patterns as Tools for Asymptotic Control

Design patterns are often taught as reusable solutions to common problems, but they can also be seen as **strategies for managing the complexity of change**. Each well-applied pattern reduces the likelihood that a single change will propagate unpredictably through the system.

For example, the **Strategy** pattern allows new behavior to be introduced via configuration or dependency injection, without modifying existing code. This aligns with the **Open/Closed Principle** and allows changes to remain ***O(1)*** as the system scales.

The **Adapter** and **Decorator** patterns reduce **immobility**, making it easier to reuse or extend functionality without altering the original implementation. This shifts reuse effort from ***O(n)*** to ***O(1)*** or ***O(k)***, where ***k*** is the number of decorators or adapters composed.

Even architectural patterns like **MVC**, **Clean Architecture**, or **Hexagonal Architecture** can be seen as attempts to **flatten the growth curve of change effort**. By isolating concerns and defining clear interfaces, they allow teams to add features without revisiting core logic—preserving long term agility.

In this sense, design patterns are not just about elegance or code reuse— they are about controlling the **structural cost of change**. When applied with intent, they form a **toolbox for asymptotic optimization**, helping teams sustain velocity as the system grows in size and complexity.

Knowing the Limits: When Big O Isn't the Right Tool

While applying Big O notation to software design offers a powerful way to reason about architectural scalability, it's important to acknowledge its limits.

Not every change in a codebase can or should be formalized asymptotically. Some edits involve exploratory work, stakeholder-driven decisions, or aesthetic preferences that don't lend themselves to quantification. Attempting to measure these with Big O adds overhead without benefit.

Moreover, Big O analysis assumes a relatively stable system boundary. In early-stage projects or rapidly evolving prototypes, optimizing for long term change complexity may be premature. In such cases, **speed of delivery** or **team velocity** may take precedence over architectural purity.

That said, the moment a system crosses a certain threshold of size, longevity, or team scale, its **change behavior becomes a bottleneck**. At that point, viewing design decisions through the lens of structural growth becomes not just useful, but essential.

The goal is not to formalize everything, but to recognize **where and when structure matters**—and to have the vocabulary to talk about it clearly.

Conclusion

Software architecture is often discussed in subjective terms: "clean," "messy," "hard to change." By introducing a structured, asymptotic

perspective, we can move beyond intuition and describe design quality with greater precision.

This approach doesn’t replace experience or craftsmanship—but it enhances them. By reasoning about architecture in terms of growth behavior, we can build systems that scale not just in features, but in their ability to adapt over time.

What's Wrong with data validation

When you don't know if you should validate

In everyday software development, many engineers find themselves asking the same question: **"Do I need to validate this data again, or can I assume it's already valid?"**

Sometimes, the answer feels uncertain. One part of the code performs validation "just in case," while another trusts the input, leading to either redundant checks or dangerous omissions. This situation creates tension between performance and safety, and often results in code that is both harder to maintain and more error-prone.

In this article, I want to examine this common problem from a design perspective and show how it's related to a deeper architectural issue: a violation of the **Liskov Substitution Principle** (LSP), one of the core tenets of object-oriented design.

But beyond that, I will propose a structural solution—based on leveraging types as **contracts of validity**—that shifts the responsibility of validation from the developer to the system itself.

The Trouble with Data Validation

When designing a system, it's common to introduce validation logic to ensure that a given data structure meets certain constraints.

Formally, we might say: we receive some input, validate that its values fall within a defined **domain of acceptable values**, and then proceed. Later in the program, the same structure may be validated again, either defensively or out of uncertainty. If the data hasn't changed, this revalidation is redundant.

While validation may impact performance, the more significant issue is ambiguity: **Who is responsible for ensuring that the data is valid at any given point in the program?**

This uncertainty creates friction in both directions:

- Some developers may **repeat validations unnecessarily**, slowing down the system and cluttering the code.
- Others may **skip validations**, wrongly assuming the input has already been checked.

Both scenarios result in fragile systems. A function may receive data that violates preconditions, not because the validation failed, but because it was silently omitted. Over time, this inconsistency becomes a source of bugs and unreliable behavior.

What seems like a minor technical detail—"just a simple check"—is actually a structural weakness in how the program models trust and correctness.

The Liskov Violation

This ambiguity around validation responsibility is more than just a code hygiene problem—it's a design flaw. More precisely, it often leads to a **violation of the Liskov Substitution Principle**

(LSP), one of the foundational ideas in object-oriented programming.

LSP states that objects of a subclass should be substitutable for objects of a superclass without altering the correctness of the program. That is, if a method expects an instance of class Parent, it should work correctly with any Child : Parent, without needing to know the exact type.

Now consider the following pattern, which may look familiar:

```
class Parent { ... }

class Child : Parent { ... }

...

// somewhere

void processValidObject(Parent parent) {
  if (parent is Child) {

    // process

  } else {

    // error

  }

}
```

This is a textbook LSP violation. The method claims to accept any Parent, but in reality, it only works if the object is a Child. The contract implied by the method signature is broken, and the type system no longer tells the truth.

In validation terms, the same mistake is often made implicitly: a

method declares that it accepts a generic structure (e.g., User, InputData, Request), but silently assumes that this structure has already passed some validation process—without enforcing it or expressing it in the type system.

As a result, invalid inputs creep into places where they were never meant to go, and the program becomes reliant on undocumented, non-local assumptions.

A Better Way: Validity Contracts and Subtypes

Instead of relying on documentation or developer discipline to enforce validation, we can model validity **explicitly in the type system**. The core idea is simple:

> ***If a certain structure can be either valid or invalid, then the type representing the "valid" version should be distinct and separate.***

For example, imagine you have a generic InputData type. Rather than passing it around and hoping it's valid, you can define a subtype—say, ValidatedInput—which represents data that has already passed all checks. The only way to obtain an instance of ValidatedInput is through a dedicated **validation function or factory**.

This creates a **contract of validity**:

- ValidatedInput guarantees invariants such as non-null fields, correct formats, and logical consistency.
- Any function that accepts ValidatedInput can trust that these preconditions are already satisfied.

- If the data is not valid, the caller cannot construct the object—the compiler will prevent misuse.

This approach shifts responsibility:

- From **“every method should validate”**
- To **“only the constructor (or factory) can validate—and once it does, the value is safe forever.”**

It’s the same principle behind concepts like non-nullable references, refined types, or units of measure. But here, we’re applying it to domain logic: using the type system to **separate raw input from trusted data**.

By doing so, we eliminate both redundant validation and unsafe assumptions. We no longer ask, “Should I validate this again?”—because the type system enforces the answer.

Real-World Examples: Files, Access, and the Danger of Assumptions

To illustrate how this principle works in practice, let’s consider a common example: file access.

Imagine a function that takes a File object and is supposed to read from it. Simple enough—but what does the File object really represent? Is it:

- A path on disk, regardless of whether it exists?
- A file that has been checked to exist?
- A file that is guaranteed to be readable?

If the type is just File, all of these interpretations are possible, and developers may start adding conditional logic like:

```
fun processFile(file: File) {
  if (file.exists() && file.canRead()) {
    // read file
  } else {
    // log or throw
  }
}
```

Once again, the function is claiming to handle any File, but actually assumes a subset of files that are already valid for reading. This leads to repeated checks, unclear responsibilities, and potentially missed edge cases.

Now consider an alternative: introduce a type ReadableFile that can only be created by a factory method like: fun tryMakeReadable(file: File): ReadableFile?

This factory encapsulates all the necessary validation (existence, permissions, etc.). Any function that receives a ReadableFile can proceed without additional checks, because **the precondition has been promoted to the type level**.

This approach scales naturally to more complex systems:

- A User that is already authenticated.
- A Document that has passed schema validation.
- A Request that has been rate-limited and sanitized.

Each of these becomes a type whose very existence **guarantees a set of invariants**, eliminating entire classes of bugs caused by misplaced trust.

Engineering Lessons and Takeaways

The problem of repeated or missing validation is not just a nuisance—it's a sign that the system doesn't express its assumptions clearly. When validity is implicit, every part of the code must remain paranoid, constantly checking and re-checking conditions that may or may not be satisfied.

By elevating **validity to a type-level concern**, we give ourselves a powerful tool:

- Functions no longer need to perform defensive checks.
- Invalid data is rejected at the boundary, not passed around inside.
- The compiler enforces correctness by construction.

This design pattern is applicable across languages and paradigms. Whether implemented through sealed classes, wrapper types, smart constructors, or dependent types, the goal is the same: **encode the program's invariants in the types themselves**.

And in doing so, we honor the Liskov Substitution Principle—not by remembering to follow it, but by making it impossible to break accidentally.

Ultimately, good software design is not just about correctness. It's

about making the right thing easy and the wrong thing hard. Using types to model validation is one of the most effective ways to achieve that goal.

Encapsulation Without Private

Rethinking Access Control

Encapsulation is one of the core pillars of object-oriented programming. It is commonly introduced through the use of access modifiers—private, protected, public, and so on—which restrict visibility of internal implementation details. Most popular object-oriented languages provide access modifiers as the default tool for enforcing encapsulation.

While this approach is effective, it tends to obscure a deeper and arguably more powerful mechanism: the use of explicit interfaces or protocols. Instead of relying on visibility constraints embedded in the language syntax, we can define behavioral contracts directly and intentionally—and often with greater precision and flexibility.

This article offers an alternative perspective on access control, suggesting that access modifiers are, in many cases, just a shorthand for declaring implicit interfaces. We'll explore how switching from implicit to explicit contracts can improve modularity, enhance design clarity, and even allow us to imagine programming languages that work without private or protected at all.

Access Modifiers as Implicit Interfaces

Access modifiers give us fine-grained control over which parts of a class are visible to the outside world. A private method is hidden from everyone except the class itself. A protected method is visible to subclasses. Package-private (in languages like Java) further

narrows or expands visibility based on the module boundary. These modifiers form a visibility matrix, controlling how internal details are exposed.

But here's the key observation: each combination of access levels effectively defines a *contract*—a set of methods that a particular group of clients is allowed to use. This is conceptually identical to an interface.

For example, when a class exposes one method as public and keeps the rest private, it's silently declaring, "This is the only thing you're allowed to call." The contract is there—it's just *invisible*.

Access modifiers are, in this sense, a mechanism for declaring *implicit* interfaces. The problem is that they are invisible to tooling and to readers. They offer no way to reason about multiple views of the same object— something explicit interfaces handle much better.

The Power of Explicit Interfaces

Interfaces (or protocols, depending on the language) let us define clearly what capabilities an object exposes—and to whom. Instead of burying access control inside the implementation using private or protected, we can expose different behaviors through multiple public interfaces. This makes the contract *visible*, *composable*, and *testable*.

Interfaces offer several advantages:

- **Clarity**: Clients only see the functionality they are supposed to

use. No hidden methods leaking through reflection or subclassing.

- **Flexibility**: Different clients can interact with the same object through different interfaces, depending on context.
- **Polymorphism**: Explicit interfaces enable runtime substitution, mocking, and dependency inversion—all core ideas in modern software design.
- **Encapsulation without obscurity**: By expressing visibility through interfaces rather than modifiers, we separate *what* is exposed from *how* it's implemented.

In effect, interfaces give us a more declarative, higher-level alternative to access modifiers. And unlike modifiers, interfaces scale well in large codebases, across modules, teams, and versions.

A Concrete Example

Let's take a simple class that uses traditional access modifiers to hide internal details:

```
public class ConsistentObject {

  public void methodA() { /* ... */ }

  protected void methodB() { /* ... */ } void
methodC() { /* ... */ } // package-private
private void methodD() { /* ... */ }

}
```

This class defines different visibility levels using Java's modifiers. But what if we rewrote the same intent using interfaces?

```
public interface IPublicConsistentObject {
  void methodA();

}

public interface IProtectedConsistentObject
extends IPublicConsistentObject {

  void methodB();

}

public interface IDefaultConsistentObject extends
IProtectedConsistentObject {

  void methodC();

}

class ConsistentObject implements

IDefaultConsistentObject {

  public void methodA() { /* ... */ }

  public void methodB() { /* ... */ }

  public void methodC() { /* ... */ }

  public void methodD() { /* ... */ }

}
```

Here, each interface builds on the previous one, modeling increasingly privileged access. The important difference is that now the contracts are **explicit and reusable**. Instead of relying on compiler-enforced visibility, we guide client interaction by choosing which interface to provide.

A consumer of this class might only receive a reference of type

IPublicConsistentObject, and therefore would only see methodA()—even though the underlying object knows how to do more.

This model makes dependencies more honest and design more modular.

The Limitation: Constructors and Instantiation

There's one area where access modifiers still feel necessary: controlling object creation.

Constructors can't be abstracted as easily as methods. You can't define a constructor in an interface, and you can't delegate instantiation to a consumer the same way you do method calls. That's why private constructors (and protected ones, in some cases) remain common— especially when enforcing factory patterns, singletons, or internal lifecycle rules.

But even this limitation is manageable.

If we embrace **factory functions** or **dependency injection**, we decouple creation from usage:

```
interface PublicAPI {
  fun doStuff()
}
private class InternalImplementation : PublicAPI {
  override fun doStuff() { /* ... */ }
}
fun createInstance(): PublicAPI {
```

```
  return InternalImplementation()
}
```

In this model, the internal class remains private, and the only way to obtain it is through the createInstance() factory. Clients never need to know—or care—how it was constructed.

This approach makes the object lifecycle part of the module's explicit API. It also aligns naturally with principles like **Inversion of Control**, and is widely used in dependency-injection frameworks.

In short, we don't need private constructors. We just need to move the responsibility of creation one level higher.

Conclusion: In Defense of Public Everything

Access modifiers have long been seen as essential to safe and clean code. But they're ultimately a **low-level mechanism** for expressing high-level ideas— contracts, roles, and boundaries.

By shifting from **implicit contracts** enforced through visibility, to **explicit contracts** expressed via interfaces and factory functions, we unlock several benefits:

- Clearer APIs
- Greater flexibility across modules
- Better separation of concerns
- Easier testing, mocking, and substitution

This doesn't mean private and protected are inherently bad. But they're a shortcut—a legacy of language design choices made

before interface-based composition became mainstream.

We can imagine a language that omits access modifiers entirely and instead relies on structured APIs to express visibility. In that world, encapsulation would still be possible—arguably even more robust—because everything would be explicit, composable, and visible by design.

And maybe that’s a world worth building toward.

Why Generics Belong in Infrastructure, Not in Architecture

Generics Are Not Magic

Generics are often praised as a cornerstone of modern type-safe programming. They allow us to abstract over types, avoid duplication, and write flexible, reusable code. In many tutorials and examples, generics are introduced as a best practice—a way to make code cleaner and "more abstract."

But in practice, many developers fall into a trap: they use generics when they don't fully understand the contracts they're creating—or failing to create. This results in code that looks flexible on the surface, but is fragile, opaque, and resistant to extension or testing. Over time, this architectural debt accumulates, and what started as an elegant abstraction becomes a barrier to maintainability.

This article is not an attack on generics. Instead, it is a reflection on how generics are commonly **misused**, especially in the context of object oriented design. It is an attempt to define boundaries: **when and where generics are appropriate**, and **where they create more harm than good**.

The main thesis is simple:

> *Generics are a low-level tool. They are powerful, but dangerous when they escape into the higher layers of application design. When used naively, they violate key*

> *principles of modularity and architecture—not because they are flawed, but because they lack enforcement of behavior.*

This article will analyze typical misuse patterns, explain how they conflict with principles like SOLID, and offer architectural alternatives that result in more robust, testable, and scalable systems.

Unclear Behavioral Contracts

One of the most common mistakes when using generics is to assume that the type parameter itself communicates enough information about the behavior of the object. But in most programming languages—including Kotlin, Java, and C#—generics only express structure, not intent.

Consider the following example:

```
fun <T> render(item: T) {
  // render the item somehow...
}
```

This function accepts any type T. But how is render supposed to know *what* to do with item? Should it call toString()? Should it access some specific field? Should it delegate to another service? The generic T does not tell us anything about the expected behavior of item.

This is where a critical architectural boundary is violated. The function **depends on an implicit assumption** about T, without enforcing that assumption. That breaks the **Liskov Substitution**

Principle (LSP): a core part of the SOLID principles. If you cannot substitute one implementation for another without risking failure, the abstraction is broken.

A better approach is to define a clear behavioral contract:

```
interface Renderable {
  fun render(): String
}
fun render(item: Renderable) {
  println(item.render())
}
```

Now the function explicitly depends on an abstraction that guarantees a specific behavior. This design is:

- easier to understand and test,
- open to extension via new implementations,
- safe to mock in unit tests,
- and resilient to future changes.

The original generic version prioritized “type flexibility.” The improved version prioritizes **semantic clarity** and **architectural integrity**.

Hidden Dependencies and Maintainability

Another common failure mode of generics is their tendency to obscure dependencies. What seems like a reusable abstraction at first glance often turns out to be a brittle and opaque structure when integrated into a real-world codebase.

Take, for example, a typical service declaration:

```
class EntityService<T> {
  fun save(entity: T) {
    // persist the entity
  }
}
```

At first, this looks generic and flexible. But what does EntityService actually depend on? How does it know how to persist an arbitrary T? What infrastructure does it use internally? Can we swap out the persistence layer? Can we mock it for tests?

The class reveals none of this. Instead, it creates a **tight coupling between the service and the type T**, without any enforced contract for how that type behaves or integrates into the system. As a result:

- Adding a new kind of entity may require changes inside the service.
- Testing becomes difficult unless you manually ensure that all T types conform to undocumented expectations.
- Extending the behavior often results in modifying generic code— which violates the **Open/Closed Principle** (OCP).

A more maintainable approach is to make the expected behavior explicit through dependency inversion:

```
interface Repository<T> {
```

```
  fun save(entity: T)
}
class InvoiceService(private val repo:
Repository<Invoice>) {
  fun process(invoice: Invoice) {
    // validate, enrich, etc.
    repo.save(invoice)
  }
}
```

Now:

- The service no longer assumes how to save an entity—it delegates.
- You can test it by injecting a fake Repository<Invoice>.
- You can reuse the logic across entities without changing the core class.
- Most importantly, the **dependencies are explicit and traceable**.

Generics alone don't provide architecture. Without a behavioral interface, a generic T is just a shape—not a contract. And shapes, unlike contracts, can't be enforced or reasoned about.

Signature Explosion and Complexity

As applications grow, so does the temptation to "scale" generic abstractions by adding more type parameters. What begins as a flexible utility can rapidly evolve into an unreadable tangle of angle

brackets.

Consider the following example:

```
class Processor<A, B, C, D>(
    private val transformer: Transformer<A, B>,
    private val validator: Validator<B, C>,
    private val handler: Handler<C, D>
) {
  fun process(input: A): D {
    val transformed = transformer.transform(input)
    val validated = validator.validate(transformed)
    return handler.handle(validated)
  }
}
```

While technically type-safe, this design is **hostile to maintenance**:

- The class signature communicates almost nothing about the domain—just structural plumbing.
- Type inference becomes hard or impossible in some contexts.
- Reading or debugging this class requires a full mental model of all four type parameters and their relationships.
- Reusability is compromised because any minor change to the pipeline may require changes across all generic bounds.

Over time, this leads to what can be described as **type signature explosion** —where understanding or modifying code requires deciphering complex generic hierarchies with no semantic

grounding.

A more readable and maintainable solution might be to **compose simpler components** with well-defined contracts:

```
interface InvoiceTransformer {

    fun transform(input: RawInvoice): ValidatedInvoice

}

interface InvoiceHandler {

    fun handle(invoice: ValidatedInvoice):
    ProcessingResult

}

class InvoiceProcessor(

    private val transformer: InvoiceTransformer,
    private val handler: InvoiceHandler

) {

    fun process(input: RawInvoice): ProcessingResult

    {

        val transformed =
        transformer.transform(input) return
        handler.handle(transformed)

    }

}
```

Here, we give up some generic reusability in exchange for **clarity, modularity, and explicit behavior**. The tradeoff favors **understandability and changeability over theoretical flexibility**.

Generics shine in isolated, well-scoped contexts. But when allowed

to govern the shape of complex application logic, they tend to obfuscate rather than clarify.

Violations of SOLID Principles

When misused, generics don't just cause local complexity—they quietly undermine the foundational principles of object-oriented design. In particular, they often lead to subtle but damaging violations of the SOLID principles, which are essential for creating maintainable and extensible software.

Let's look at three key violations: Liskov Substitution Principle (LSP) The LSP states that objects of a superclass (or interface) should be replaceable with objects of a subclass (or implementation) **without breaking the behavior of the program**.

When you write something like:

```
fun <T> process(item: T) {
  item.doSomething() // ← illegal unless T has a constraint
}
```

You're assuming `T` supports some behavior—but that's not enforced. The compiler won't stop someone from passing in a type that breaks your assumption. There's no way to substitute safely. You've created an illusion of abstraction without any behavioral contract.

By contrast:

```
fun process(item: Processable) {
```

```
  item.doSomething()
}
```

…enforces a clear substitution rule, making the code safer and more robust.

Dependency Inversion Principle (DIP)

DIP advises that high-level modules should not depend on low-level details, but on abstractions.

Generic classes often invert this principle by requiring the caller to provide the low-level implementation directly—e.g., passing in a specific `T`, instead of relying on an abstract interface.

```
class Processor<T> {
  fun execute(input: T) { ... }
}
```

This class depends entirely on a concrete `T` that must be passed from outside. There's no abstraction being injected, no clear boundary of responsibility. It's a structural dependency, not a behavioral one.

Open/Closed Principle (OCP)

The OCP states that software entities should be **open for extension but closed for modification**.

When generics are used without abstraction, any new behavior or special case often requires **modifying the generic code**:

```
class Exporter<T> {
  fun export(item: T) {
    if (item is SpecialType) {
      exportSpecial(item)
    } else {
      exportDefault(item)
    }
  }
}
```

This is a classic anti-pattern: the type parameter is not expressive enough to capture the variation, so runtime type checks leak in, and the generic code becomes the dumping ground for exceptions.

Proper use of interfaces avoids this entirely:

```
interface Exportable {
  fun export(): String
}
```

Now new behaviors are introduced by implementing a contract—not by editing old code.

The Illusion of Convenience

At first glance, generics seem like the ultimate convenience feature. They reduce duplication, allow for reusable components, and let developers defer decisions about specific types. For many, this feels like a win—the code compiles, it's flexible, and it works for

“everything.”

But this convenience is often **an illusion**—a short-term gain that masks long-term complexity.

Generics make code compile, even when it’s conceptually broken

A generic function like:

```
fun <T> handle(data: T) {
  // do something with data
}
```

will happily compile with any type T. But the compiler doesn’t know—and cannot know—whether T makes sense in this context. You’re not expressing **intent**, only **possibility**.

The result is code that’s syntactically valid, but semantically vague. It’s not clear what handle() is supposed to do or for whom it is designed. The contract is missing—and so is the guarantee.

Short-term flexibility leads to long-term rigidity Ironically, the more generic your code becomes, the harder it is to change.

- Adding a new edge case often requires editing the generic itself.
- Understanding behavior requires mentally tracing through multiple type parameters.
- Testing becomes fragile because the same code behaves differently depending on what T happens to be.

This is the **inverse of good design**. Instead of isolating change, you centralize it. Instead of making code self-descriptive, you rely on documentation (if any) to explain expectations.

Cognitive overhead increases—without architectural benefits

Generic heavy code is harder to read, harder to teach, and harder to debug. It sacrifices clarity in the name of theoretical reuse. And often, the reuse never materializes—the generic ends up being used in only one or two places, each time with a slightly different constraint that the original author never anticipated.

By contrast, **well-named interfaces communicate behavior** at a glance and reduce the burden on future readers and maintainers.

When Generics Actually Make Sense

Despite the risks of misuse, generics do have their place—and when used appropriately, they provide significant value. The key is to understand **where** they belong: in low-level utilities, infrastructure code, and data containers, not in business logic or domain architecture.

Use Case 1: Collections and Low-Level Utilities

Collections are the textbook example of safe and effective generic use. Lists, maps, queues —all benefit from type safety without

prescribing behavior.

```
val list: List<User> = listOf(user1, user2)
```

There's no behavioral expectation on User here. The collection is a data container—not a logic executor. This is where generics shine: flexible storage, no assumptions.

Use Case 2: Type-Safe Builders and DSLs

Generic types can help create fluent, expressive APIs without boilerplate.

```
inline fun <reified T> inject(): T = ...
```

Dependency injection frameworks, serialization libraries, and query builders use generics to infer types at compile time while hiding internal complexity. The key is that these generic operations are **scoped and localized**—they don't leak into the domain logic or impose behavioral expectations.

Use Case 3: Infrastructure Components

Inside frameworks like Retrofit, Room, or MapStruct, generics are used to avoid repetitive code across many types:

```
interface ApiService {
  @GET("items")
  fun <T> getItems(): Call<List<T>> }
```

Again, this is fine because:

- It’s not part of your domain model.
- It doesn’t enforce or imply behavior.
- It stays in the infrastructure layer, isolated from business logic.

Use Case 4: Composition Within a Closed Context

Generics can work when all the types involved are well understood and the boundaries are tightly controlled—for example, internal caching, object pooling, or orchestration layers that are never exposed to the rest of the system.

Pattern: Generics should generalize data—not behavior

This is the simplest test:

> ***Does your generic represent “what the object is,” or “what the object does”?***

If the answer is behavior, an interface is almost always a better fit. Interfaces define expectations. Generics do not.

Design Principle: Abstract Behavior, Not Types

At the heart of robust software design is a simple rule: **abstract behavior, not concrete types.**

This principle is often violated when developers reach for generics too early—attempting to write “type-agnostic” code without clearly defining *what the code is supposed to do*. But good architecture is not about hiding variation—it’s about expressing **intent** explicitly and enabling safe extension.

What generics hide, interfaces reveal

Consider this function:

```
fun <T> process(data: T) { ... }
```

Now compare it with:

```
fun process(data: Processable) { ... }
```

The first version is flexible—it accepts anything—but says nothing about what behavior is expected from T. The second version imposes a contract. It communicates to the reader, the tester, and the compiler: “I expect something that knows how to process itself.”

Behavior lives in interfaces—and that’s a good thing

Behavior is the backbone of design. If your code depends on T having a method like validate() or render(), then your function does not work for all T—it only works for types that support those behaviors.

Trying to express this via generics often leads to brittle bounds, unclear constraints, and runtime surprises.

Instead, define a behavioral abstraction:

```
interface Validatable {
  fun validate(): Boolean
}
```

Then build your system on top of that contract. This approach:

- documents the expected behavior,

- enables static guarantees,
- supports dependency inversion and testing,
- and keeps your domain logic clean and intention revealing.

Don’t abstract over types just to avoid repetition

Too often, developers write generic code simply to avoid writing the same function twice— even when the functions do *different things*. This kind of DRY (Don’t Repeat Yourself) obsession leads to **leaky abstractions** that save lines of code at the expense of clarity and flexibility.

Instead of abstracting too soon, **duplicate with intent**, and only unify abstractions when a common contract naturally emerges.

Practical Guidelines for Using Generics

Generics can be used effectively—but only within well-defined constraints. To help ensure they serve your architecture rather than undermine it, here are several actionable guidelines.

1. Use generics only when behavior is irrelevant

If your code needs *no knowledge* of how an object behaves—it only stores it, passes it along, or wraps it—generics may be appropriate.

Examples:

- Caching mechanisms
- Data containers (e.g. List<T>, Box<T>)

- Mapping functions that don’t inspect behavior But as soon as logic depends on what the object *does*, you should stop and ask: **Should this be an interface instead?**

2. Encapsulate generics within infrastructure layers

Keep generics hidden within internal or infrastructural components like:

- networking wrappers,
- serializers,
- DI containers,
- repositories.

Once you enter business logic—services, use cases, domain models— **prefer explicit behavioral contracts**.

3. Prefer interfaces to generic constraints

Instead of:

```
fun <T: Validatable> validate(item: T) { ... }
```

Write:

```
fun validate(item: Validatable) { ... }
```

Both work, but the second is simpler, clearer, and easier to extend. Avoid overcomplicating function signatures when the goal is to express *intent*, not polymorphic overengineering.

4. Never use generics to bypass proper modeling

If you're using generics because you're unsure what behavior your types should expose, it's a red flag. You're likely **postponing a modeling decision** that needs to be made now, not hidden behind T.

5. Audit generic-heavy code for testability and clarity

If a class or function has multiple type parameters and unclear constraints, ask:

- Can I write a meaningful unit test for this?
- Can someone else read and understand what it does?
- Can I swap out one type for another without breaking things?

If the answer is no, your generics are probably doing too much.

6. Generalize through composition, not over parameterization

Favor patterns like Strategy, Adapter, and Decorator when introducing variation in behavior. These patterns are **explicit, readable, and testable**— unlike chains of generic functions with hidden logic.

Rule of Thumb

> *If you can't explain what a generic function does without*
>
> *referencing its type parameter, you probably shouldn't be using one.*

Conclusion: Generics Are a Low-Level Tool

Generics are a powerful feature. They can reduce duplication, increase type safety, and enable elegant abstractions—**when used deliberately and with clear constraints**. But too often, they are applied as a shortcut: a way to make code "flexible" without carefully defining how that code should behave.

When generics escape their appropriate boundaries—especially into high level business logic or architectural components—they tend to:

- obscure intent,
- violate key design principles like SOLID,
- reduce testability,
- and increase the cognitive burden on everyone who works with the code.

This is not because generics are "bad." It's because **they lack opinion**. They allow anything—and good architecture requires boundaries.

Think of generics like goto or null: They are valid constructs. But they carry **semantic risk**—they give you freedom without guardrails. And just like goto, when used inappropriately, they bypass structure rather than support it.

Design Guideline

> ***Use generics when you need to generalize over data. Use interfaces when you need to generalize***

over behavior.

Put generics in infrastructure. Keep them out of your domain. Don’t abstract what you haven’t modeled. And don’t rely on syntax to save you from design decisions.

Well-designed systems are not those with the fewest lines of code or the most flexibility. They are the ones where **responsibilities are clear, change is safe, and behavior is explicit.**

Designing a Flexible Ability System for Games

Why Ability Systems Must Be Flexible

In game development, the ability system is often one of the most demanding components in terms of flexibility. At the design stage, it's nearly impossible to predict what spells, abilities, or skills will exist in the final version—or what updates will introduce in the future.

This article is about how I approached this uncertainty by abstracting the process of executing abilities.

At its core, an ability is nothing more than a set of actions. A minimalistic ability interface might consist of a single method like apply(). But in practice, things are rarely that simple. The complexity doesn't lie in calling the ability—it lies in **determining whether the ability can or should be used at all**.

In order to manage this complexity and allow for future expansion, we need a flexible, modular approach that decouples ability execution from the conditions under which it may or may not proceed. This leads us to rethink how to structure ability logic from the ground up.

The First Layer: Ability Checks as Chainable Components

Every ability begins with a series of checks that determine whether

it can be used. These checks are usually things like:

- Is the ability off cooldown?
- Does the character have enough mana?
- Is the target within range?

Right away, it becomes obvious that **not every ability needs every check**. For instance, some abilities might not require mana, or may be usable at any distance.

This means different abilities require different sets of preconditions. However, **many of these checks are reused** across multiple abilities. Cooldown, mana, and range checks are common across dozens of spells. If these checks are duplicated everywhere, any change to their logic must be applied in many places—creating fragility.

To avoid duplication and enable flexibility, we can **extract each check into its own object** implementing a shared interface. Then, we link them

together in a single, ordered chain.

This is the classic **Chain of Responsibility** pattern.

Here's what such an interface might look like:

```
interface CastChecker {
  CastChecker nextChecker { get; set; }
  bool check();
```

```
}
```

And here's an example of a simple chain: CooldownChecker → ManaChecker → CastRangeChecker Each checker performs a specific validation and, if successful, passes control to the next in the chain.

This structure allows for reuse, recombination, and centralized changes— the foundation of a truly flexible system.

Executing the Chain: Sequential Validation and Error Handling

Once we've assembled a chain of CastChecker objects, the system can process them sequentially to validate whether an ability can be used.

Each checker in the chain follows the same logic:

1. If its own condition fails, it **stops the chain** and reports an error (e.g. "Not enough mana").
2. If the condition passes, it **calls the next checker**, continuing the validation process.

Here's a simple implementation outline:

```
bool CastChecker.check() {
  if (!thisConditionIsMet()) {
    showErrorMessageToPlayer(); return false;
  } else if (nextChecker != null) {
    return nextChecker.check(); } else {
```

```
    return true;
  }
}
```

This design introduces a few key benefits:

1. Composable and Maintainable Checks

You can build a custom validation pipeline per ability without rewriting shared logic. For example:

- A fireball might need mana, cooldown, and range.
- A healing spell might only need cooldown and line of sight.

2. Readable Flow

Since each check is self-contained, its logic stays focused and understandable. The CastChecker interface allows adding new conditions without modifying existing ones.

3. Centralized Error Handling

Each checker can report its own failure reason—giving clear, targeted feedback to the player.

This modularity is what sets the system apart from ad hoc validation logic. We're no longer writing giant if statements or switch-cases. Instead, we assemble abilities like LEGO blocks—combining reusable, testable pieces.

Abstraction via SkillCastRequest

Now that we've covered how to validate an ability using a chain of checkers, we need to think about how the ability actually gets executed—and more importantly, how to represent that execution as an abstract, independent process.

Let's introduce a new interface: SkillCastRequest.

This interface doesn't care whether the ability is an instant fireball or a multi-phase ritual. It simply represents "a request to perform an action," and exposes a standard way to start or cancel it:

```
interface SkillCastRequest {
  void startRequest();
  void cancelRequest();
}
```

This abstraction lets us treat the execution logic as a **first-class citizen** in our architecture.

Instead of having every ability directly embed its own complex execution logic (animations, delays, input windows, etc.), we separate that into a reusable request object.

Benefits of this approach:

- **Reusability:** The same request logic (e.g., a charging bar or input sequence) can be used for multiple skills.
- **Interruptibility:** Requests can be paused, canceled, or restarted independently from the ability system.

- **Asynchronicity:** Since startRequest() doesn't return anything, it can easily support coroutine-like or event-driven flows.

In essence, this abstraction decouples **what the skill does** from **how it gets initiated**—a critical distinction for building flexible gameplay systems.

TerminalChecker and Executing the Skill

We now have two powerful tools in our toolbox:

- A chain of **CastCheckers** that validates whether a skill can be used.
- A **SkillCastRequest** that encapsulates the process of executing that skill.

But how do we **tie them together** in a way that guarantees execution only happens if all checks pass?

That's where the **TerminalChecker** comes in.

It's a special node in the chain—always placed at the end—whose job is to trigger the actual startRequest() call when all prior checks succeed.

Example:

```
class TerminalChecker implements CastChecker {
    CastChecker nextChecker = null;
    SkillCastRequest request;
      bool check() {
```

```
            request.startRequest();
            return true;
        }
}
```

In a full chain, it might look like this: CooldownChecker → ManaChecker → RangeChecker → TerminalChecker

Only if the first three validations pass will the request begin.

Why separate the final execution?

1. **Keeps responsibilities clean.** Each checker only checks; only the final node triggers execution.
2. **Easier to reuse.** You can create different TerminalCheckers for different types of execution (e.g., networked requests, instant local effects, delayed effects).
3. **Supports asynchronous operations.** For example, some skills might involve charging, targeting, or waiting for input before resolving. The request object can handle that without polluting the checker logic.

This final step **bridges the gap** between *should the ability run* and *go ahead and run it.*

Binding the Skill and the Request

We've now split ability logic into two distinct domains:

- **Validation logic**—handled by the CastChecker chain
- **Execution logic**—encapsulated in a SkillCastRequest

But how do we represent an actual **skill**—something the player can activate?

Simple: we bind both parts together under a unified interface.

Defining the Skill interface:

```
interface Skill {
  string name;
  SkillCastRequest request;
  CastChecker checker;
  bool cast() {
     return checker.check();
  }
}
```

When the player tries to use a skill:

1. The cast() method is called.
2. The checker chain is executed.
3. If the final TerminalChecker is reached, it starts the SkillCastRequest.

This design gives us **complete separation of concerns**:

- The ability's name and metadata live in the Skill object.
- Validation logic lives in its checker chain.
- Execution logic lives in the request.

Why this is powerful:

- You can reuse checkers and requests across multiple skills.
- You can dynamically assemble or swap out parts at runtime.
- You can subclass or wrap Skill objects to add logging, cooldown tracking, analytics, or multiplayer synchronization—without changing the base structure.

This turns your skills into **pure data + behavior composition**, making them ideal for designers, modders, and procedural generation.

Example and Conclusion: A Universal Execution Framework

Let's put it all together with a concrete example: the TeleportationSkill.

Teleportation is a perfect case because it breaks common assumptions:

- It **doesn't** require mana.
- It **can't** be used in combat.
- It **requires the player to stand on a teleportation pad**.
- It has a long cooldown.
- It must **wait for the player to confirm** the destination.

Using our architecture, this complex behavior is no

problem.

We assemble it like this:

Checkers:

- CooldownChecker
- InCombatChecker (custom logic: player must be out of combat)
- SurfaceChecker (verifies player is on correct surface)
- TerminalChecker (starts the request)

Requests:

- TeleportationRequest, which:
    a. Opens a destination selection UI
    b. Waits for confirmation
    c. Moves the character
- Skill object:

```
Skill teleport = new Skill(
  name = "Teleport",
  checker = new CooldownChecker(
      next = new InCombatChecker(
      next = new SurfaceChecker(
          next = new TerminalChecker(
          request = teleportationRequest)
      )
  )
```

```
  ),
  request = teleportationRequest
);
```

This entire skill is **fully declarative and composable**. No tight coupling, no duplicated logic. If we later want to use the same teleportation behavior for enemies or items—we just plug in the same request.

Final Thoughts

By separating validation, execution, and composition:

- We gain **modularity**: each component is testable and replaceable.
- We gain **extensibility**: adding new checks or execution styles is trivial.
- We gain **clarity**: game logic becomes declarative, not imperative.

Summary Diagram

Skill

├── name: "Teleport"

├── checker:

```
│ └── CooldownChecker
│     └── InCombatChecker
│         └── SurfaceChecker
│             └── TerminalChecker → request.startRequest()
└── request: TeleportationRequest
```

This is a universal framework not just for spells or attacks, but for **any game mechanic** where action depends on conditions.

You can use this to build skill trees, item usage systems, interaction mechanics—anything where “can I do this?” must be evaluated before “do this.”

TYPE-SAFE POLYMORPHIC CONSTRUCTORS VIA COMPILE-TIME GUARANTEES

In many programming systems, there is a recurring need to instantiate objects from raw binary data. This is especially common in protocols or serialization layers where object reconstruction must follow a predictable contract. While object-oriented languages typically offer powerful polymorphic behavior through interfaces and virtual methods, constructors are notably not polymorphic. This limitation becomes apparent when you want to enforce that all subclasses of a given protocol or interface must implement a constructor with a specific signature—such as accepting a byte[] and an index.

The issue stems from the fact that virtual dispatch mechanisms rely on the existence of an already constructed object. Without an instance, the runtime has no way of determining which method implementation to invoke. As a result, constructors cannot be declared in interfaces or abstract base classes, and the compiler cannot enforce constructor conformance across all subclasses.

However, this limitation does not invalidate the desire to express such constraints. In fact, it signals a deeper need: developers often wish to encode architectural contracts into their type system. They want to guarantee, at compile time, that all types conforming to a

protocol can be constructed from raw data. But without a dedicated language feature, this kind of contract enforcement must be emulated through design patterns and discipline.

This article explores how to achieve such compile-time guarantees using static constructs—specifically enums, fixed-size function arrays, and interface patterns. We'll build a technique that offers the flexibility of polymorphic construction while maintaining type safety and performance. Along the way, we'll uncover how this approach aligns with the Open Closed Principle and can scale across large, extensible systems.

Protocols as Constraints and the Single Responsibility Principle

In statically typed languages, interfaces and abstract classes (collectively referred to here as protocols) serve as tools for defining polymorphic behavior. They describe what an object can do, enabling inversion of control and decoupling between components. However, developers often push these protocols

beyond their original purpose—not just to describe behavior, but also to constrain the structure of the codebase.

For example, one might wish to ensure that all implementations of a protocol include a constructor with a specific signature, or that they are registered in a central factory. Since protocols cannot express such meta level requirements directly, developers embed these constraints informally in documentation or enforce them via code reviews. But this approach is fragile and prone to errors,

especially as a system grows.

This leads to a violation of the **Single Responsibility Principle (SRP)**. A protocol intended to define polymorphic behavior ends up serving a second role: enforcing architectural discipline. This dual responsibility increases the cognitive load on developers and introduces risks when adding new subclasses.

What this suggests is that modern languages might benefit from a new, declarative construct: one explicitly designed for constraining code structure rather than just behavior. Until such features exist, developers are left to simulate them through alternative mechanisms. In the next section, we'll explore one such mechanism—building a safe and scalable factory system that approximates "virtual constructors" through static dispatch.

Static Dispatch and Virtual Constructors

To simulate polymorphic construction in a type-safe and performant way, we can use a combination of static data structures and reflection. The idea is simple: for each type that conforms to a protocol, we register a factory function that knows how to construct it from raw bytes.

Let's take a closer look at a real-world implementation in C#. The system defines a factory that reads a skillIndex from a byte array, looks up a constructor for the corresponding type, and invokes it using reflection. To avoid repeated reflection overhead, constructors are compiled once and cached in a static array:

```
static class SkillFactory {

        delegate ISkill SkillConstructorSignature(byte[]
bytes, ref int index);


        static readonly SkillConstructorSignature[]
cachedConstructors =

            new
SkillConstructorSignature[(int)SkillType.SkillsCount];


        public static ISkill fromBytes(byte[] bytes, ref
int index)

        {

            var skillIndex =
(int)(SkillType)BufferUtils.readI2(bytes, ref index);


            if (cachedConstructors[skillIndex] != null)

            {

                return
cachedConstructors[skillIndex].Invoke(bytes, ref index);

            }


            var type = SkillData.classes[skillIndex];

            var constructor = type.GetConstructor(new[]

            {

                typeof(byte[]).MakeArrayType(),

                typeof(int).MakeByRefType()

            });
```

```
            if (constructor == null)
            {
                throw new Exception($"No constructor from
raw bytes for type {type}");
            }
            var newExpression =
Expression.New(constructor);
            var compiledExpression = Expression

.Lambda<SkillConstructorSignature>(newExpression)
                .Compile();
            cachedConstructors[skillIndex] =
compiledExpression;
            return compiledExpression.Invoke(bytes, ref
index);
        }
        public static void toBytes(ISkill skill, byte[]
bytes, ref int index)
        {
            BufferUtils.writeI2((short)skill.type, bytes,
ref index);
            skill.serialize(bytes, ref index);
        }
    }
```

This approach emulates a **virtual constructor table**, where cachedConstructors acts as a dispatch vector. Each function in

the array knows how to build a specific subtype from raw data. When a new skill type is added:

1. It gets assigned a unique enum value in SkillType.
2. A corresponding class is added to SkillData.classes.
3. Its constructor with signature (byte[], ref int) is implemented.

If all three are done correctly, the system "just works." If something is missing—like a constructor with the wrong signature—the code will fail fast and clearly at runtime. And with some compile-time scaffolding, we can catch these issues even earlier, as we'll explore in the next section.

Compile-Time Control via Static Structures

One of the most common pitfalls in large, extensible systems is forgetting to update some central logic when adding a new type. For example, when a new subtype is introduced, a developer might forget to update a factory method or register the new class—leading to runtime bugs that are hard to trace. Ideally, the compiler would help us catch such mistakes.

Some languages, like Swift or Rust, enforce exhaustiveness in switch statements. If you forget to handle a new enum case, the compiler will flag it. Unfortunately, mainstream languages like C#, C++, and TypeScript do not enforce such checks by default.

To work around this, we can encode the exhaustiveness requirement into the type system using a simple trick: by reserving a final enum case— commonly named Count, TypesCount, or Last—and using it to size a static array. Here's how this works:

```
enum SkillType {
  Fireball,
  IceBlast,
  Heal,
  // ...
  SkillsCount
}
```

Now we can create an array with a fixed size based on SkillsCount:

```
static readonly SkillConstructorSignature[]
cachedConstructors = new
SkillConstructorSignature[(int)SkillType.SkillsCount
];
```

This enforces an implicit contract: if you forget to add a factory function for a new SkillType, the missing index in the array becomes a null slot, leading to a controlled failure at runtime. But more importantly, the structure of the code naturally forces the developer to “touch all the places” whenever a new type is added. You cannot forget to extend the array without breaking the program.

This is not just a runtime safety mechanism—it’s a **static assertion embedded in the shape of the code**. And since enum values are typically assigned incrementally from 0, array indexing becomes safe and efficient.

While it’s true that arrays don’t enforce ordering, and mistakes like index mismatches are possible, in practice such bugs are rare, easy

to localize, and trivial to fix. Surprisingly, despite their simplicity, constructs like these are underutilized or outright unavailable in many languages that favor reflection or runtime registration.

In the next section, we'll compare this approach with interfaces and explore when to use static dispatch, enums, or full-blown polymorphism.

Interfaces vs Enums: When to Use What

At first glance, enums, arrays, and interfaces may appear to serve different purposes. But under the hood, they often enable similar patterns: dispatch, constraint enforcement, and structural safety. Choosing between them depends not only on performance or readability, but on the **type of control** you want the compiler to enforce.

Let's break this down:

Enums and Static Arrays

Best suited for **data-centric dispatch**. For example:

- You have a closed set of cases.
- You want the compiler to help ensure completeness (e.g., via array size or exhaustive switch).
- You're controlling external behavior, like parsing byte streams or serializing formats.

They shine when you want to associate behavior or data with a finite set of symbolic cases—especially when that set changes rarely and predictably.

Interfaces and Polymorphism

Best suited for **instructional logic**, encapsulation, and extension. For example:

- You want behavior defined per type, co-located with its data.
- You want to respect the Open-Closed Principle: new behavior through new classes, not edits to central logic.
- You're building systems where responsibilities are distributed across modules or plugins.

They shine when types evolve independently, and when grouping logic with data improves clarity.

Gray Area: Polymorphic Static Data

Sometimes, you want each type to expose a **static property**, such as a type tag or category label. These values are shared across instances and differ between types. For example:

```
interface ISkill {
  val type: SkillType
}
```

Here, the type acts like a **"polymorphic static"** value: it behaves like static data, but it's accessed through instance-level polymorphism. This is often the only viable approach in languages that don't support true type-level functions or compile-time constants per subclass.

Composability

Interestingly, both approaches can and often **should** coexist. For example:

- The enum SkillType can be used to drive deserialization.
- The interface ISkill defines shared behavior and serialization logic.
- A static array links SkillType to constructor functions.
- The type getter on ISkill ensures round-trip consistency between the deserialized type and the serialized form.

This hybrid approach results in code that is **extensible**, **type-safe**, and **compilable under constraints**—a rare trifecta in dynamic factory systems.

Reflection and Performance

One of the main concerns with reflection-based solutions is performance. Traditional reflection—calling GetConstructor() and Invoke()—is notoriously slow, especially in tight loops or real-time systems. However, modern runtimes offer efficient alternatives by leveraging **expression trees** or **dynamic code generation**.

In our implementation, we use System.Linq.Expressions to create and compile constructor delegates at runtime. This means the reflection step only happens once per type. The resulting compiled delegate is cached in a static array and reused for all future instantiations. This brings us **close to raw constructor performance**, with the flexibility of dynamic dispatch.

Here's what makes this approach production-grade:

- **First-use compilation**: Constructors are compiled only once per type.
- **Zero-cost dispatch after caching**: After caching, constructor calls are as fast as manually written lambdas.
- **Memory-safe**: The array index (backed by an enum) ensures no accidental out-of-bounds access.
- **Controlled failure**: If a constructor with the expected signature is missing, the system fails early and visibly.

This technique gives you **the best of both worlds**: dynamic extensibility with static-like performance. More importantly, it respects the Open-Closed Principle: new subclasses can be added without modifying the factory, provided they follow the expected constructor contract.

From the consumer's point of view, constructing a new ISkill from a byte stream is a one-liner. But behind that one line lies a rigorous, testable, and efficient system of type-safe dispatch—one that avoids runtime conditionals, switch statements, or unchecked reflection.

Conclusion

By combining enums, static arrays, interface patterns, and compile-time invariants, we've shown that it's possible to emulate **polymorphic constructors** in statically typed languages—without sacrificing performance or type safety. This approach offers:

- **Scalability**: Easily accommodates dozens or hundreds of

subtypes without increasing complexity.

- **Maintainability**: Changes in one subtype do not require modifications to the core logic.
- **Safety**: Design violations are caught at compile time or on first use, not in production.
- **Performance**: Cached expression trees deliver constructor calls as fast as native methods.

More importantly, this technique aligns closely with core software architecture principles:

- It enforces **Open-Closed Principle (OCP)**: systems can be extended with new types without altering existing code.
- It encourages **Separation of Concerns**: type creation, serialization, and logic are cleanly modularized.
- It allows developers to **encode structural contracts**—even when the language lacks native support for things like constructor constraints or static assertions.

In many ways, this pattern elevates the type system from a passive safety net to an active design enforcer. It nudges the compiler toward the role of a co-architect: not just catching errors, but shaping the way systems grow.

USING CLOSURES TO EXTEND CLASS BEHAVIOR WITHOUT VIOLATING INTERFACE BOUNDARIES

The Hidden Cost of Interface Changes

In software architecture, one of the most subtle forms of technical debt comes not from what we write—but from what we expose.

It often starts with a small, well-meaning change: we want to add access to a private field or internal method of an important class. But we only need that access in one very specific client—a test, a legacy integration, a specialized handler. And so we think: maybe we should just extend the interface a little?

But this decision, while easy to justify in the moment, has long-lasting consequences. Every extension to an interface creates a new public

contract. And public contracts are sticky. Other developers start to rely on it. The internal detail becomes external behavior. You've added surface area— and lost encapsulation.

This article explores a practical architectural technique to solve such cases elegantly: using **closures** to provide internal access *without* exposing internal state, and without bloating your interface. We'll walk through the problem, naive solutions, and then land on a

design pattern that preserves encapsulation, keeps dependencies clean, and minimizes ripple effects in your codebase.

Let's dive in.

The Problem: One Class, One Secret, One Needy Client

Imagine you're working with a critical class in your codebase—let's call it ImportantService. It's well-designed, well-tested, and widely used across the system. Somewhere deep inside, it holds a private detail— say, a cache map or a connection state—that is intentionally hidden from the outside world.

Now suppose you have a very specific client—maybe a migration job, a diagnostic tool, or an adapter—that needs to interact with that internal detail just once, in one specific place. It doesn't need full control over the class. It doesn't even need to know how the field works. It just needs to use it once, responsibly.

But there's a problem: that field isn't accessible. The only way to reach it is by modifying ImportantService's public interface. And that feels wrong.

You hesitate, and rightfully so. Because modifying a public interface means opening a door that was previously closed. And once that door is open, it rarely gets closed again. Even if your intention was to use it *just once*, you can't stop other parts of the codebase from wandering through it later.

So you're stuck:

- You **don't** want to break encapsulation.

- You **do** need access in one place.
- And you're **not** sure how to solve this cleanly.

That's the problem we're going to tackle.

Naïve Solution: Just Expose the Field

The most straightforward way to solve this is deceptively simple:

> *Just add a new method to the public interface.*

For example, you might write:

```
class ImportantService {
  private val cache = mutableMapOf<String,
String>()
  fun getCache(): MutableMap<String, String> {
    return cache
  }
}
```

Now your client can interact with the internal cache freely. Problem solved? Not quite.

You've just violated the principle of encapsulation—and done so permanently. What was once an implementation detail is now a public API. Even if your intention was to use this method in only one place, you've effectively told every other developer: "Hey, feel free to reach into this internal structure whenever you like."

This creates three major problems:

1. **Fragility**—The internal structure can no longer change freely. If you swap cache for a different mechanism later, you'll break all clients relying on getCache().
2. **Misuse Risk**—Other parts of the codebase might start using the exposed method in ways you didn't anticipate—maybe even mutating shared state.
3. **Interface Pollution**—The class's public interface becomes bloated with methods that serve niche needs, making it harder to understand and harder to maintain.

This approach works in the short term but almost always causes regret later. It's a technical shortcut with long-term cost.

And most importantly: it introduces a contract where none was meant to exist. The moment you expose something, you own it—forever.

Better But Clumsy: Interface Splitting and Casting

If exposing a private field directly is too risky, a slightly better solution might be to split the interface. That is, define two interfaces:

- A **public interface** that contains only the safe, general-purpose methods used by the majority of the system.
- A **specialized interface** that includes access to the internal detail— used only where truly needed.

It could look like this:

```
interface PublicAPI {
    fun doSomething()
```

```
}
interface InternalAPI : PublicAPI {
    fun getCache(): MutableMap<String, String>
}
class ImportantService : InternalAPI {
    private val cache = mutableMapOf<String,
    String>()
    override fun doSomething() {
        // ...
    }
    override fun getCache():
    MutableMap<String, String> = cache
}
```

In your codebase, you expose ImportantService only as PublicAPI. But in the one place that needs getCache(), you can do a cast: val service: PublicAPI = getImportantService() val internal = service as? InternalAPI internal?.getCache()?.put("debug", "value") This approach does preserve encapsulation *to an extent*—the getCache() method isn't visible unless you explicitly cast to InternalAPI.

But it comes with its own set of problems:

1. **Unsafe Type Casting**—Even though you "know" the type at runtime, you're opting out of compile-time safety. That's a slippery slope and a code smell in most modern systems.

2. **Leaky Abstractions**—You've now introduced multiple views of the same object. If other developers find and misuse the InternalAPI, you're back to polluting your architecture.
3. **Refactoring Overhead**—You've complicated your class hierarchy and added extra interfaces to maintain, all to solve a one-off case.

In theory, interface segregation is a clean architectural practice. But in this scenario—where the need is rare, isolated, and very specific—it can feel like overkill. Worse, the as? cast becomes a tacit admission: "We broke the type system a little, but it's probably fine."

There must be a better way.

Elegant Trick: Using Closures to Encapsulate Access

Instead of modifying the interface or relying on unsafe casts, we can flip the problem:

> *What if we don't expose the internal detail, but instead expose a controlled interaction with it?*

This is where **closures** (or lambdas) come in.

Rather than letting the client access the internal field, we let the *internal field access the client logic*—but only in a very narrow and safe way.

Let's take the same ImportantService, but this time, we give it a method that accepts a function:

```
class ImportantService {

    private val cache = mutableMapOf<String,
    String>()

    fun <R> withCache(action: (MutableMap<String,
    String>) -> R): R {

        return action(cache)

    }

}
```

Now in the client:

```
val service = ImportantService() service.withCache
{ cache -> cache["debug"] = "value" }
```

This looks similar to accessing the cache directly, but it's a different animal architecturally.

Why is this better?

1. **Controlled Exposure**

   The internal detail is still private. You're only giving clients a *momentary, scoped interaction* with it—and only when you choose to.

2. **No Interface Pollution**

   You didn't add a new getter. You didn't expand your public API. The class's surface area remains tight.

3. **No Type Leaks**

   No casting, no subclassing, no interface proliferation.

Everything is encapsulated within the boundary of the method.

4. **Clear Intent**

   By naming the method withCache, you signal that this is a controlled and intentional handoff—not an invitation to poke around.

Bonus: Functional Composition

You can combine this pattern with other functional tools—mapping, chaining, filtering—to do expressive work with internal state without ever revealing it.

This is already a much cleaner solution. But it has a hidden cost we'll explore next—and it involves **dependency direction**.

But Wait: Inverted Dependencies

At first glance, the closure based approach seems perfect: it keeps your internal field private, avoids interface pollution, and provides clients with just enough power to get their job done.

But there's a subtle shift happening here—one that could quietly violate a key architectural principle: **dependency direction**.

Let's look again at the method:

```
fun <R> withCache(action:
(MutableMap<String, String>) -> R): R {
  return action(cache)
}
```

Who's in control here?

- The **client** defines the action, i.e. the logic that operates on the cache.
- The **ImportantService** receives and executes that logic.

This means the core class—the thing we want to keep stable and authoritative—is now **depending on a function provided by an external client**.

That's an **inversion of control**. And not the good kind.

In traditional architecture, **the stable component should not depend on the volatile one**. That's the essence of the *Stable Dependencies Principle*. And here, we just flipped that—the ImportantService now calls out to a client-defined function.

Why is that a problem?

1. **Hidden Coupling**—Now the core class indirectly relies on logic defined elsewhere. This creates temporal and behavioral coupling that's hard to trace.
2. **Change Fragility**—If the client's closure changes in a way that breaks assumptions, it can affect the core class, even if its own code hasn't changed.
3. **Harder to Test in Isolation**—Testing the core class now requires mocking or simulating the injected closure logic, which might not be desirable.

So while closures give us **encapsulation**, they come at the cost of **reversing the dependency graph**.

To keep things clean, we need to restore the direction—without losing the flexibility. And that's exactly what we'll tackle next.

The Final Abstraction: A Dedicated Closure Carrier

To resolve the dependency inversion without giving up the elegance of closures, we introduce a new architectural element: *a small, dedicated abstraction that acts as a neutral carrier of logic.*

Instead of passing the closure from the client *directly* into the ImportantService, we wrap that logic inside a purpose-built object—let's call it CacheAction.

```
fun interface CacheAction<R> {
  fun execute(cache: MutableMap<String, String>): R
}
```

This interface does one thing: defines a contract for interacting with the internal cache. It is not the client, and it is not the service—it's an abstraction between them.

Now, the ImportantService depends on CacheAction, not on an arbitrary client-provided lambda:

```
class ImportantService {
    private val cache = mutableMapOf<String, String>()
    fun <R> perform(action: CacheAction<R>): R {
        return action.execute(cache)
    }
```

```
}
```

And on the client side:

```
val action = CacheAction<String> {

    cache -> cache["debug"] = "value"

    "done"

}

val result = service.perform(action)
```

Why is this better?

1. **Dependency Flow is Restored** The service depends only on the abstraction (CacheAction), which is stable and controlled. It doesn't care where the implementation comes from.
2. **Encapsulation is Preserved** The cache is still private. No getters, no leaks.
3. **Interface is Clean and Intention-Revealing** The method name perform + the typed parameter make the intent explicit: you're performing an action *on behalf of* the service, using its private parts.
4. **Testability is Improved** You can now test ImportantService with stub CacheActions, or test different CacheAction implementations independently.
5. **Optional Reuse and Registry** In larger systems, you can even register reusable CacheActions—or compose them—giving you plugin-like extensibility without ever exposing internals.

This pattern gives you the power of closures, the safety of clean dependencies, and the expressiveness of functional composition—all while keeping your architecture tidy.

We've now solved the original problem *without* polluting the interface, *without* unsafe casts, and *without* inverting control in the wrong direction.

Let's wrap it up.

Summary: Extend Without Breaking Let's recap the journey.

We started with a simple but frustrating problem: How do you give a specific client access to a private part of a class without compromising the design of the system?

We explored several options:

1. **Exposing a getter**—quick but harmful. It breaks encapsulation and pollutes the interface.
2. **Splitting interfaces and casting**—safer in theory, but clunky and error-prone in practice.
3. **Using closures**—elegant and concise, but introduces inverted dependencies.
4. **Introducing a dedicated abstraction**—the best of all worlds: encapsulated behavior, clean dependency flow, and composability.

When to use this pattern

This isn't a tool for every situation. But it's a great fit when:

- You need to access internal state in a **very limited scope**.
- You want to **avoid expanding the public API**.
- You care about **preserving architectural boundaries** and avoiding coupling.
- You prefer **explicit contracts** over ad hoc conventions.

It's especially useful in large codebases or long-lived systems, where interface cleanliness and dependency direction have real consequences.

Closing thought

Architecture is about trade-offs. But sometimes, with a bit of creativity, we can have our cake and eat it too—extending functionality without breaking design, hiding complexity without losing power, and building systems that are both flexible and principled.

Closures, when used thoughtfully, are one of those quiet superpowers.

REFLECTION IS NOT THE ENEMY OF TYPE SAFETY

When Type Safety Meets Reality

In modern software development, type systems are often seen as bastions of safety. They catch our mistakes, enforce structure, and give us confidence that our programs behave as intended. But what happens when our perfectly typed world collides with the messy reality of external data—JSON payloads, third-party APIs, or loosely typed config files?

In these situations, we are forced to bridge two incompatible worlds: the safe, static world of types and the chaotic, dynamic world of data. And to do that, we need tools that let us inspect and transform untyped inputs into typed structures. We need reflection.

Surprisingly, reflection—often criticized as a "type system backdoor"— can, in fact, serve as a powerful ally in maintaining type safety. Used correctly, it doesn't undermine our type system. It reinforces it.

The Paradox of Reflection

Reflection is often treated as a necessary evil—a loophole that allows developers to peek behind the curtain of the type system and manipulate objects in unsafe ways. In many programming cultures, it carries a stigma: "If you need reflection, you probably did something

wrong.”

But this view is simplistic. Reflection, by itself, is not inherently unsafe. The problem arises only when it’s used without discipline.

When reflection is constrained by types—when we use it not to circumvent the type system, but to uphold it at runtime—it becomes a tool of enforcement, not evasion. In that sense, reflection is not the antithesis of type safety, but its extension into runtime.

In statically typed languages, the type system protects us during compilation. But after deserialization, or when dealing with Any or Object, that safety disappears unless we have runtime mechanisms to restore it.

Reflection, when implemented carefully, allows us to restore that safety in a structured way.

TypeScript’s Structural Blind Spot

These reflections came into sharp focus when I revisited TypeScript’s object model. TypeScript is often praised for its powerful type system, but it’s important to remember: its types are erased at runtime. What you write in your code editor exists only at compile time—by the time your code runs, the types are gone.

This leads to a frustrating limitation: in TypeScript, you cannot reflect on generic types. You can write function<T>(value: T), but you have no way of examining T at runtime—its structure, its fields, whether it conforms to an interface. There's no object is T because T is not real

at runtime. It's transparent.

This makes a lot of sense in a language like C++, where runtime performance and binary size are critical. But in TypeScript, which already runs on a virtual machine and bears the full runtime overhead of JavaScript, the absence of type metadata feels self-defeating. The runtime is already heavy. What's a few more singleton metadata entries per type?

This design choice limits our ability to write reusable, type-safe data processing code. And ironically, it leads to more any, not less.

Why object is T Matters

Imagine receiving a blob of untyped data— say, a JSON object from an external API. You want to validate it, parse it, and then pass it safely into your well-typed domain model. In many languages, this involves a simple and expressive check: if (object is T).

But in TypeScript, this is impossible in the general case. You cannot check whether an object conforms to a generic type T, because T doesn't exist at runtime. Instead, you must write tedious manual guards: test that each expected field exists and has the right type. It's boilerplate-heavy and error prone.

This becomes especially frustrating when you want to write reusable code. Suppose you're building a deserialization utility: it should take an unknown

object and try to coerce it into some known type T. You can write this elegantly in languages that support runtime type information. In TypeScript, it becomes a hacky dance of hardcoded property checks, with no guarantees and lots of duplication.

The result? Either you abandon strict typing at the edges of your system, or you write custom guards for every possible type—neither of which scales well.

The Runtime Philosophy: C++ vs. TypeScript

To be fair, TypeScript's avoidance of runtime metadata isn't accidental. It follows a conscious design philosophy: types are a developer aid, not a runtime feature. The type system exists purely to improve the development experience—it vanishes during compilation. This choice minimizes the impact on JavaScript runtime performance and bundle size.

In contrast, languages like C++ make a different tradeoff. C++ assumes you're working with raw data—bytes, structs, hardware memory layouts. Runtime reflection in C++ would be both expensive and dangerous. It breaks assumptions about performance and control. That's why C++ generally avoids it, or limits it to niche tools like RTTI or template metaprogramming.

But TypeScript is not C++. It doesn't run on bare metal. It runs inside a dynamic, reflective, garbage-collected virtual machine—JavaScript. This environment already has built-in metadata, object shape introspection, prototype chains, dynamic dispatch.

That's what makes the TypeScript runtime philosophy feel contradictory. It pretends to be lean like C++, while sitting atop a bloated JavaScript engine. If you're already paying for a heavyweight runtime, why not let the type system leave behind some breadcrumbs to make that runtime safer?

Reflection as a Type-Safe Bridge

There's a common assumption that reflection always undermines type safety. But that's only true when reflection is unbounded—when you access fields by string names, cast values blindly, or modify private state without checks.

But what if we constrain reflection within the type system itself?

In some languages, this is already the norm. For example:

- In **Scala**, you can use typeclasses or macros to derive serializers and parsers in a type-safe way.
- In **Rust**, serde uses custom derive macros to generate safe serialization code based on compile-time metadata.
- In **Kotlin**, libraries like kotlinx.serialization can safely serialize and deserialize objects using compiler-generated type metadata.
- Even **C#** allows safe reflection via expression trees and strongly-typed generic builders.

In these ecosystems, reflection isn't a hack—it's a language feature. And more importantly, it becomes the bridge between untyped inputs and strongly typed models.

TypeScript could follow a similar path. Instead of erasing types entirely, it could optionally emit lightweight metadata for selected interfaces or classes. Just enough to make object is T a reality. Just enough to make the edge between dynamic and static smooth, not jagged.

Toward a Smarter Runtime

Type safety shouldn't stop at the compiler boundary. For modern systems that rely heavily on data interchange, serialization, and runtime composition, we need the ability to reason about types even after the code has shipped. Otherwise, we're forced to rebuild a type system at runtime—manually, painfully, and often incorrectly.

TypeScript has made incredible strides in improving developer experience, but its runtime story is still lacking. By acknowledging that type information can be both optional and useful at runtime, the language could open the door to safer, more expressive patterns—without abandoning its lightweight roots.

Reflection doesn't have to mean dynamic chaos. Done right, it becomes a controlled gateway—a validator, a contract enforcer, a safe lens through which raw data becomes structured meaning.

And in a language that proudly calls itself *Type*-Script, that doesn't sound like a betrayal of its principles. It sounds like their natural evolution.

The Compiler Is My Co-Author

The Paradox of Unexpected Joy

There's a unique kind of joy that only developers experience. It's not the launch day, not user praise, and not peer recognition. It's the moment you hit the build button—and it just works. It works sooner than you expected. It works when you were still mentally preparing for hours of debugging.

That feeling of sudden completeness, as if a door you've been pushing against quietly opened by itself, is rare—but unforgettable. It's one of the quiet motivators that keeps us coming back to the codebase, even after long nights of red error messages and false starts.

The Strategy: Think Globally, Act Locally

When facing a large, messy problem, I do something simple: I pick the nearest subtask and focus solely on that. I intentionally avoid holding the full complexity in my head—not out of laziness, but because I've already thought through the big picture. The direction is clear; now it's just execution.

This is the only way I've found to make meaningful progress without drowning in analysis. Once the architecture is understood, the rest becomes, quite literally, a matter of technique. It's not blind coding. It's local focus, grounded in global reasoning.

Refactoring as Flow

Not long ago, I had to extract shared functionality from four nearly identical classes. A common protocol was clear, and the solution was to introduce an abstract base class. It was the kind of task any engineer would file under “routine.” A couple of hours at most.

But something unexpected happened: it became **fun**. Every time I successfully lifted a method out into the base class and deleted a redundant piece of code, I felt a small surge of satisfaction. It was like clearing clutter off a desk—you don’t notice how messy it was until it’s clean.

It became so engaging that I stopped thinking about the new features I was supposed to implement. I just wanted to finish the cleanup.

Into the Red: A Willing Descent

Before starting the refactoring, I deliberately broke the codebase. Not in a reckless way, but strategically—I made changes that rendered the program uncompilable, knowing full well it wouldn’t build until the entire transformation was complete.

It was a conscious dive into the storm. The IDE lit up with red squiggles. Every file I opened screamed for attention. But this wasn’t chaos; it was progress in disguise. Each red error wasn’t a failure—it was a placeholder, a signpost saying: “You’re halfway there.”

There’s something deeply motivating about working through that

blizzard of errors. It's like chopping your way through a dense forest—you don't stop until you see daylight.

That Moment of Silence

And then, suddenly, the red was gone.

I pressed the build button one last time—not expecting much, just checking how far I'd come. But this time… it compiled. No errors. No warnings. Just clean output.

I launched the app, clicked around, poked at the edges, looking for something I had missed. Some broken button. A forgotten case. But everything was working. The functionality I meant to add was already there —ready to go.

It wasn't magic. It was just… done. And that quiet, matter-of-fact *"done"* is one of the most satisfying things engineering can give you.

The Compiler Is Not Your Enemy

We often treat the compiler like a gatekeeper, an obstacle standing between us and a working program. But moments like this remind me: the compiler is not the enemy. It's a partner. A strict one, yes—but one that holds the line when you're tempted to cut corners.

The sea of red wasn't a punishment. It was a map. It told me exactly what needed to change, and in what order. It kept me honest. It made sure the system was coherent **before** it ran, not after something blew up in production.

In a world full of uncertainty, the compiler is one of the few allies that demands correctness up front. And when it finally goes silent, it’s not just a relief—it’s a sign that the system is whole again.