Traits and Mixins: A Comparative Analysis of Two Modern Compositional Patterns

Mason Bott

University of Colorado Boulder

Abstract

In this paper, we present a comparative analysis of ‘traits‘ in the Rust programming language and ‘mixins‘ in the Dart programming language. We evaluate their distinct approaches to code reuse and polymorphism, contrasting Rust's contract-based philosophy with Dart's implementation-based model. Furthermore, we analyze the architectural implications of these patterns, specifically examining their impact on type safety, conflict resolution, and software testability.

1 Introduction

For decades, classical inheritance has been the cornerstone of object-oriented programming (OOP), offering a powerful mechanism for code reuse and polymorphism. While classical inheritance set a precedence for ideation within object-oriented languages, it had problems such as rigid class hierarchies that left something to be desired. In response, modern programming languages have increasingly favored more flexible compositional patterns. This paper investigates two prominent examples of this shift: traits, as implemented in Rust, and mixins, as found in Dart. Although both features provide alternatives to classical inheritance, they are rooted in distinct design philosophies and have significant, differing impacts on software architecture. This analysis compares their mechanisms, their solutions to long-standing inheritance problems, and their alignment with the design goals of their respective languages; Rust as a systems language and Dart as a UI-focused application-level language.

1.1 Motivation

Classical inheritance, while foundational to OOP, has long been associated with significant architectural challenges. Issues such as the ”fragile base class problem” (Mikhajlov and Sekerinski 1998), where changes in a base class can unexpectedly break derived classes, and the tight coupling it creates between a class and its superclass, lead to rigid and brittle designs. Furthermore, languages that attempt to solve these issues with multiple inheritance often introduce new complexities, most notably the ”diamond problem” (Black, Schärli, and Ducasse 2002), which creates ambiguity in how to resolve inherited members from a common ancestor.

The presence of these problems prompted a paradigm shift in the philosophy of software design famously articulated as the principle of ”composition over inheritance” (Gamma et al. 1994). This philosophy advocates for building complex functionality by assembling smaller, independent objects rather than inheriting from a monolithic base class. Modern language designers have responded by introducing new features that enable composition as a first-class concept, moving beyond the limitations of classical inheritance while preserving the benefits of code reuse and polymorphism. This paper is motivated by the need to analyze two of the most prominent, yet divergent, modern approaches: traits and mixins. By providing a direct comparison, we can better understand their respective solutions to these long-standing inheritance problems and the new architectural trade-offs they introduce.

1.2 Background and Related Work

The evolution of object-oriented programming has involved a continuous search for effective code reuse and polymorphism. Early approaches centered on single inheritance, but its limitations quickly led to divergent solutions. Languages like C++ embraced ”multiple inheritance” (Stroustrup 1989), offering great flexibility but also bringing in the diamond problem mentioned in §1.1.

In response, languages like Java and C# championed interfaces to separate contractual guarantees from implementation. This model, often paired with abstract classes for sharing implementation within a single-inheritance tree, provided a safer alternative. However, these purely abstract interfaces created a significant ”interface evolution problem.” As Goetz noted in a formalization on the issues of Java interfaces, ”once published, it is impossible to add methods to an interface without breaking existing implementations” (Goetz and Darcy 2010). Java solved this problem with default methods in Java 8, allowing interfaces to provide concrete behavior.

As a direct response to the limitations of both single and multiple inheritance, new compositional models were proposed. The concept of ”mixins” was formalized as a mechanism for implementation sharing through a predictable, linearized inheritance chain (Bracha and Cook 1990). Similarly, the concept of ”traits” was formally introduced as a ”tiling module” of atomic behaviors that could be composed without the rigid hierarchies of classical inheritance (Black, Schärli, and Ducasse 2002). These two concepts form the foundation of the modern compositional patterns explored

in this paper.

2 A Dive into Traits and Mixins

To appreciate how these two features provide alternatives to classical inheritance, we must first understand their distinct mechanics. This section will deconstruct Rust's trait system and Dart's mixin system, examining the specific language features, design philosophies, and core capabilities that define each approach.

2.1 Rust's Trait System

At its core, Rust's trait system serves as the language's primary model for abstraction and polymorphism, radically supplanting the classical class-based inheritance found in most object-oriented languages. Conceptually, a trait defines a set of behaviors or capabilities that a type can implement. They can be understood as "interfaces with default implementations," a design that directly addresses the "interface evolution problem" (as noted in §1.2) in a way similar to Java 8's default methods. A type can explicitly opt into a trait like so:

```
impl SomeTrait for SomeType {
  // Required methods from the trait
  // must be defined here, or the code
  // will not compile.
  fn some_method(&self) -> String {
    String::from("Some string")
  }
}
```

Polymorphism Through traits, Rust provides powerful and flexible polymorphism by the use of two distinct mechanisms. The first is static dispatch via *trait bounds*. This mechanism uses generics to define functions that accept any type as long as it implements a specific trait's contract. For example, a function can be bounded to only accept types that implement the `Drawable` trait:

```
fn draw_item<T: Drawable>(item: &T) {
  item.draw(); // Compile-time call
}
```

The compiler resolves these generic bounds at compile-time through monomorphization (The Rust Project Developers 2018c). This means the compiler generates a specialized, non-generic version of the function for each concrete type it is actually called with, replacing the generic trait type parameter with that specific type. This results in abstractions with performance identical to manually specialized code at the cost of larger binaries.

The second is dynamic dispatch via *trait objects*, which provides runtime polymorphism. This enables the use of heterogeneous collections where the concrete types are unknown until runtime (Klabnik and Nichols 2019). Naturally, this includes a small cost of a vtable lookup to resolve the method call:

```
// Assume item1 and item2 are defined
let items: Vec<&dyn Drawable> =
    vec![&item1, &item2];

for item in &items {
  item.draw(); // Call via vtable
}
```

This dual-dispatch system is a direct expression of Rust's philosophy as a systems language. Static dispatch via trait bounds is the default, fulfilling the "zero-cost abstraction" promise: compile-time polymorphism with performance identical to hand-written, specialized code. Dynamic dispatch via trait objects is an explicit, opt-in choice for when runtime flexibility is required (like heterogeneous collections), with its cost (a vtable lookup) made clear. This allows developers to make deliberate, fine-grained performance trade-offs, rather than enforcing a single "one-size-fits-all" model.

Associated Types Traits extend their contractual power beyond methods by including "associated types" (The Rust Project Developers 2018a). This allows a trait to define a generic "family" of types that are related to the implementing type, a common requirement for abstractions like iterators or collections. This makes the contract far more expressive, as it can define relationships *between* types, not just required behaviors. For example, a generic `Iterator` trait can define its contract not just by the `next` method, but also by the types it operates on:

```
trait Iterator {
  type Item; // The iteratee type

  fn next(&mut self) -> Option<Self::
      Item>;
}
```

Any implementer must define what `Item` type they produce. For example, a counter that yields integers:

```
struct Counter {
  count: u32,
}

impl Iterator for Counter {
  // Specifies the concrete type
  //
  // Note that omitting this results in
  // a compilation error.
  type Item = u32;

  fn next(&mut self)
        -> Option<Self::Item> {
    self.count += 1;
    Some(self.count)
  }
}
```

Supertraits Rust also supports something close to inheritance called Supertraits (The Rust Project Developers 2018b), which allows for the creation of more complex contracts by composing simpler ones. This is not classical implementation inheritance; rather, it is a sub-typing relationship on the *contracts themselves*. A trait can specify that it requires another trait to be implemented:

```
trait Drawable {
  fn draw(&self);
}

// Clickable "inherits" from Drawable
trait Clickable: Drawable {
  fn on_click(&self);
}
```

This declaration states that any type implementing `Clickable` must *also* implement `Drawable`. This allows for building layered, more specific abstractions. Crucially, the implementing type still provides all the concrete logic (or uses default methods); it inherits the contractual requirements, not the implementation or state from a parent class. This reinforces the trait system's role as a purely compositional, contract-based model.

2.2 Dart's Mixin System

In contrast to Rust's trait system, Dart's mixins are fundamentally a mechanism for implementation reuse rather than contractual abstraction (Bracha 2015). Conceptually, a mixin is a "class" from which another class can borrow implementation. This borrowing includes concrete methods and instance fields without being its subclass. A class can explicitly compose in any number of mixins using the `with` keyword:

```
class SomeClass with MixinA, MixinB {
  // ... class implementation
}
```

This design is defined by several key features, including its handling of state, its conflict resolution strategy, and its use in polymorphism.

Stateful Composition and Linearization Unlike Rust traits or traditional Java interfaces, Dart mixins are not limited to methods; they can contain concrete instance variables. When a class uses a mixin, it absorbs this state and behavior as if it were defined directly in the class itself.

This approach creates a potential for "diamond problem" conflicts, which Dart resolves through mixin linearization. When a class uses `with` to apply multiple mixins, Dart creates a single, linear inheritance chain. Methods in later mixins simply override those in earlier ones, providing a simple, predictable resolution path. For example, expanding upon the `SomeClass` example from above...

```
mixin MixinA {
  // defines `mixinMethod`
  String mixinMethod() {
    return "Called from MixinA!";
  }
}

mixin MixinB {
  // defines `mixinMethod` with
  // identical signature as `MixinA`
  String mixinMethod() {
    return "Called from MixinB!";
  }
}
```

When the method `mixinMethod()` is called on an instance of `SomeClass`, since `MixinB` was the last mixin applied to `SomeClass`, the result will be that of what `MixinB` defined for the method.

Type Safety via Constraints Mixins can be applied to any class, but they often need to call methods on the class they are mixed into. To do this safely, Dart provides the `on` keyword, which acts as a constraint restricting a mixin's application only to classes that extend or implement a specific superclass or interface. This mechanism is explored in depth in §3.3, where we contrast it with Rust's approach to compositional type safety.

```
mixin Turbo on Engine {
  // Safely calls Engine's method
  void boost() => ignite();
}
```

Mixins as Polymorphic Types While mixins are a tool for implementation sharing, they also create a new polymorphic type. When `SomeClass` uses `with SomeMixin`, the `SomeClass` object *is also* of type `SomeMixin`. This allows for polymorphism based on mixin application, which is a powerful abstraction tool in its own right:

```
mixin Clickable {
  void onClick() => print("Clicked!");
}

class Button with Clickable {}
class Page with Clickable {}

// Polymorphism: this function accepts
// any object that uses `Clickable`
void triggerClick(Clickable item) {
  item.onClick();
}

// triggerClick(Button()); // Works
// triggerClick(Page());   // Works
```

Use Case: The Flutter Framework This pattern is heavily utilized in Dart's primary application framework, Flutter, to share discrete behaviors among 'Widget' states. A canonical example is the `SingleTickerProviderStateMixin`, which provides the complex logic to schedule animations (The Flutter Team 2025a). Rather than forcing a complex class hierarchy, a 'State' class simply applies the mixin:

```
class _SomeState extends State<MyWidget>
    with SingleTickerProviderStateMixin
        {
  // Class now has Ticker functionality
}
```

This injects the necessary state and methods for ticker functionality directly into the state, exemplifying the pragmatic, compositional approach mixins enable for UI development.

3 Comparative Analysis of the Compositional Patterns

While both traits and mixins provide powerful alternatives to classical inheritance, their underlying philosophies and mechanisms lead to distinctly different architectural outcomes. This section analyzes these differences, beginning with their core conceptual purpose and moving to their solutions for inheritance problems, their approach to polymorphism, and their integration with their respective type systems.

3.1 Conceptual Distinction: Contract vs. Implementation

The primary distinction lies in their conceptual purpose: Rust's traits are fundamentally a contract, defining what a type can do, while Dart's mixins are a mechanism for implementation sharing, defining how a behavior is provided. A Rust type *implements* a trait, promising to fulfill its abstract interface, even if some methods have default implementations. A Dart class *uses* a mixin to directly absorb its concrete fields and methods into its own definition. Both champion the "composition over inheritance" principle (Gamma et al. 1994), but they do so from opposing starting points.

Consider a simple `Logger` capability. In Rust, this is naturally expressed as a contract that a type agrees to fulfill:

```
1  // The trait defines a contract
2  trait Logger {
3    fn log(&self, msg: &str);
4  }
5
6  struct SomeService {
7    service_id: u32,
8  }
9
10 // The struct implements the contract
11 impl Logger for SomeService {
12   fn log(&self, msg: &str) {
13     println!("[Svc {}] {}", self.
           service_id, msg);
14   }
15 }
```

The `Logger` trait defines an abstract behavior, and `SomeService` must provide its own concrete implementation. The trait itself holds no state and provides no implementation (though it could provide a default).

In Dart, the same goal is achieved by injecting implementation, including state. The mixin is a bundle of reusable functionality:

```
1  mixin Logger {
2    // Mixins can contain concrete state
3    String logPrefix = "DEFAULT";
4    // ...and concrete methods
5    void log(String msg) {
6      print("[$logPrefix] $msg");
7    }
8  }
9
10 // The class absorbs the implementation
11 class SomeService with Logger {
12   SomeService() {
13     logPrefix = "SERVICE";
14   }
15 }
```

Here, `SomeService` does not implement `Logger`; it *composes in* its concrete state (`logPrefix`) and its concrete method (`log`). This exemplifies the core difference: Rust traits are an "opt-in" for a capability, while Dart mixins are an "include" of functionality.

3.2 Resolution of the Diamond Problem

Both systems elegantly solve the diamond problem mentioned in §1.1, but their solutions diverge based on their core philosophy.

We saw in §2.2 that Dart employs the canonical **mixin linearization**, an implementation expressed by Bracha in an early formalization of mixins where mixins "depend upon linearization to place them in an appropriate location in the inheritance chain" (Bracha and Cook 1990). If multiple mixins provide a method with the same name, the method from the mixin applied last in the `with` clause simply overrides any previous ones.

```
1  mixin Walker {
2    String move() => "Walking";
3  }
4
5  mixin Flyer {
6    String move() => "Flying";
7  }
8
9  // Flyer.move() "wins" as it's last
10 class Duck with Walker, Flyer {}
11
12 void main() { print(Duck().move()); }
13 // Output: Flying
```

Rust avoids the problem entirely because a trait's methods are **namespaced** to the trait itself. A type can implement two different traits that both define a method with the same signature, and there is no ambiguity; the caller must explicitly state which trait's method they are invoking.

```
1  trait Walker {
2    fn move(&self) -> &'static str {
3      "Walking"
4    }
5  }
6
7  trait Flyer {
8    fn move(&self) -> &'static str {
9      "Flying"
10   }
11 }
12
13 struct Duck;
14 impl Walker for Duck {}
15 impl Flyer for Duck {}
16
17 fn main() {
18   let duck = Duck;
19   // No ambiguity; must be explicit
20   println!("{}", Walker::move(&duck));
```

```
    println!("{}", Flyer::move(&duck));
}
// Output: Walking
//         Flying
```

This demonstrates Dart's pragmatic preference for a simple, linear override chain versus Rust's insistence on explicit disambiguation, which prevents conflicts from arising in the first place and requires no special resolution rules.

3.3 A Comparison on Type Safety

While both languages provide strong, static type systems, the way traits and mixins integrate with those systems reveals their different design priorities. Rust's trait system is deeply interwoven with its core goal of providing compile-time guarantees for memory and concurrency safety. This is most evident in its use of marker traits (traits with no methods) like `Send` and `Sync`.

A type is `Send` if it is safe to move to another thread, and `Sync` if it is safe to be shared (via reference) between threads. These are not just optional features; they are contracts that the compiler understands and enforces. The type system uses these trait bounds to statically prevent data races before the program can even run (Jung et al. 2021). Attempting to send a non-`Send` type (like `std::rc::Rc`, a reference-counted pointer) across a thread boundary results in a hard compile-time error.

```
use std::rc::Rc;
use std::thread;

fn main() {
  // Rc<T> is not Send
  let data = Rc::new("not sendable");

  // Fails to compile
  thread::spawn(move || {
    println!("{}", data);
  });
}
// Compiler Error: `Rc<&str>` cannot be
// sent between threads safely
```

Dart's mixin system, in contrast, is not concerned with low-level memory or concurrency safety, as Dart is a garbage-collected language that manages concurrency via isolates. Instead, its type safety mechanism focuses on ensuring the structural integrity of its compositions. As seen in §2.2, the `on` keyword acts as a precondition, guaranteeing that a mixin is only applied to a class that extends a required superclass. This ensures that any methods the mixin calls from its "host" (via `this`) are guaranteed to exist, preventing runtime errors.

```
class Engine {
  void ignite() => print("Engine on");
}

// Mixin only to Engine subclasses
mixin Turbo on Engine {
  void boost() {
    ignite(); // Safe to call this
    print("Boost engaged!");
  }
}

// OK: Car is a subclass of Engine
class Car extends Engine with Turbo {}

class Chair {}

// Error: Chair does not extend Engine
class RocketChair extends Chair with
    Turbo {}
```

This comparison clearly shows the different philosophies: Rust uses traits to enforce deep, system-level safety guarantees, while Dart uses mixin constraints to ensure application-level, compositional safety.

3.4 Testability and Mocking

The contract-based nature of Rust traits facilitates a powerful testing pattern: mock implementations. Because a trait defines an abstract contract, test code can provide a "test double" or "mock" implementation to isolate the unit under test from its dependencies.

Consider a `Consumer` struct that is generic over a `Behavior` trait:

```
// A simple contractual trait
trait Behavior {
  fn operation(&self, value: u32);
}

// The unit under test, generic over the
// contract
struct Consumer<B: Behavior> {
  dependency: B,
}

impl<B: Behavior> Consumer<B> {
  // The method we want to test
  fn run(&self, input: u32) {
    // It uses the dependency's
    // operation
    self.dependency.operation(input);
  }
}
```

In order to test the internal dependency's `Behavior` operation, we provide a `MockBehavior` that uses interior mutability (`RefCell`) to record that its `operation` method was called. While strictly an implementation detail, `RefCell` is necessary here for our mock since the `operation` method on `Behavior` takes in an immutable reference `&self`.

```
// The mock implementation
struct MockBehavior {
  // We wrap in RefCell as this
  // allows mutation via &self
  calls: RefCell<Vec<u32>>,
}

impl Behavior for MockBehavior {
  fn operation(&self, value: u32) {
    // Record the interaction
    self.calls.borrow_mut().push(value);
```

```
  }
}

// In Rust, #[test] is the standard
// way to declare we are creating a
// test function.
#[test]
fn test_consumer_calls_behavior() {
  // Create the mock
  let mock = MockBehavior {
    calls: RefCell::new(Vec::new()),
  };

  // Inject it
  let consumer = Consumer {
    dependency: mock
  };

  // Run the method under test
  consumer.run(42);

  // Verify mock was called as expected
  let calls = consumer.dependency.calls;
  assert_eq!(calls.borrow().len(), 1);
  assert_eq!(calls.borrow()[0], 42);
}
```

This pattern allows the test to verify the interaction that `run` correctly calls `operation` in complete isolation. This is fundamental to unit testing, and is so essential that the Rust ecosystem includes libraries like `mockall` that can automatically generate such mock implementations from trait definitions (KodeKloud).

Dart's mixins, being concrete implementations rather than abstract contracts, present different testing considerations. Because mixins inject behavior directly into a class, they are typically tested either in isolation by creating a minimal test harness class, or by testing the composed class as a whole. For self-contained mixins, isolation testing is straightforward:

```
mixin ValidationMixin {
  String? validateEmail(String email) {
    return email.contains('@') ? null :
        'Invalid email';
  }
}

// Minimal test harness
class _TestHarness with ValidationMixin
    {}

void main() {
  test('ValidationMixin validates email'
      , () {
    final harness = _TestHarness();
    expect(harness.validateEmail('
        invalid'),
           equals('Invalid email'));
  });
}
```

However, when mixins depend on their host class via the `on` keyword, testing requires providing a concrete implementation of the required superclass. This can be done manually by creating a mock version of the superclass that implements its abstract methods and records interactions.

```
// The required superclass contract
abstract class Identity {
  String get identifier;
}

// The mixin we want to test, which is
// dependent on Identity
mixin Logger on Identity {
  // Calls the superclass's 'identifier'
  // getter
  void log(String message) {
    print("[$identifier] $message");
  }
}

// A manual mock implementation of the
// superclass
class MockIdentity implements Identity {
  int identifierCallCount = 0;

  @override
  String get identifier {
    identifierCallCount++;
    return 'MOCK_ID';
  }
}

// The test harness extends the mock and
// mixes in the behavior
class TestHarness extends MockIdentity
    with Logger {}
```

This `TestHarness` class satisfies the mixin's dependency and allows the test to verify the interaction. The test can now assert that the superclass's `identifier` getter was called by the mixin's `log` method.

```
void main() {
  test('Logger mixin calls identifier
      getter', () {
    // Create the harness
    final harness = TestHarness();

    // Call the mixed-in method
    harness.log("Test message");

    // Verify the superclass getter was
    // called
    expect(
      harness.identifierCallCount, 1
    );
  });
}
```

This manual pattern is fundamental to isolating the mixin. Fortunately, the process of creating the harness class (both `MockIdentity` and `TestHarness`) can be automated by libraries like `mockito`, which generate mock classes from abstract definitions (The Flutter Team 2025b).

Overall, this comparison reveals a fundamental trade-off: Rust's contract-based traits make dependency injection

and mocking straightforward by design, as the abstraction boundary is explicit. Dart's implementation-based mixins require more elaborate test harnesses when dependencies are involved, though they offer simpler testing when mixins are self-contained.

4 Conclusion

This comparative analysis reveals that Rust's traits and Dart's mixins represent fundamentally distinct solutions to the same problem: the limitations of classical inheritance. Traits embody a contract-first philosophy, where types opt into abstract capabilities, enabling zero-cost polymorphism and compiler-enforced safety guarantees through mechanisms like marker traits. Mixins, conversely, embody an implementation-first philosophy, where classes compose concrete, stateful behaviors through a pragmatic linearization strategy.

These philosophical differences manifest in concrete architectural trade-offs. Traits excel in systems programming contexts where compile-time optimization and memory safety are paramount. Rust's `Send` and `Sync` traits demonstrate how compositional patterns can encode deep invariants that the type system can verify. Mixins thrive in application development, particularly UI frameworks like Flutter, where the ability to inject complex, stateful behaviors (animations, lifecycle management) across diverse components without inheritance hierarchies proves invaluable.

Both systems successfully resolve long-standing inheritance problems like the diamond problem, the fragile base class problem, and the interface evolution problem, but through markedly different mechanisms. Rust's explicit disambiguation and namespace separation prioritize clarity and compile-time reasoning, while Dart's linearization and `on` constraints prioritize developer ergonomics and rapid composition.

The choice between these patterns ultimately depends on domain requirements. For systems-level programming where performance, memory safety, and explicit control are paramount, trait-based composition provides the necessary guarantees without runtime overhead. For application development where rapid iteration, stateful composition, and ergonomic code sharing are priorities, mixin-based composition offers pragmatic flexibility. These are not competing solutions but complementary approaches, each optimized for its respective domain.

Critically, both patterns validate the "composition over inheritance" principle not merely as philosophy but as practical language design. As software systems grow in complexity and languages continue to specialize, we expect future languages to adopt similar compositional patterns tailored to their specific domains. The success of traits and mixins suggests that the evolution of object-oriented programming has yet to find a single best-agreed-upon compositional model and it is likely that model will never come. As of now, the solution lies not in discovering a single "correct" compositional model, but in recognizing that different computational domains demand different compositional strategies.

References

Black, A. P.; Schärli, N.; and Ducasse, S. 2002. Traits: Composable Units of Behavior. Technical Report $\mathrm{TR}_{C}SE_{0}2 - 012, PortlandStateUniversity$.

Bracha, G. 2015. *The Dart Programming Language*. Addison-Wesley Professional. ISBN 978-0321927705.

Bracha, G.; and Cook, W. R. 1990. Mixin-based Inheritance. In *OOPSLA/ECOOP '90 Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*.

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Goetz, B.; and Darcy, J. 2010. Interface evolution via "public defender" methods. https://cr.openjdk.org/~briangoetz/lambda/Defender%20Methods%20v3.pdf. Accessed: 2025-11-04.

Jung, R.; Jourdan, J.-H.; Krebbers, R.; and Dreyer, D. 2021. Safe Systems Programming in Rust. *Communications of the ACM*, 64(4): 144–152.

Klabnik, S.; and Nichols, C. 2019. *The Rust Programming Language*. No Starch Press.

KodeKloud. ???? Mocking Dependencies in Rust. https://notes.kodekloud.com/docs/Rust-Programming/Testing-Continuous-Integration/Mocking-Dependencies-in-Rust. Accessed: 2025-11-08.

Mikhajlov, L.; and Sekerinski, E. 1998. A study of the fragile base class problem. In *ECOOP'98–Object-Oriented Programming: 12th European Conference Brussels, Belgium, July 20–24, 1998 Proceedings*, 355–382. Springer.

Stroustrup, B. 1989. Multiple inheritance for C++. *Computing Systems*, 2(4): 367–395. Revised version of paper presented at EUUG Spring 1987 Conference.

The Flutter Team. 2025a. Architectural overview. https://docs.flutter.dev/resources/architectural-overview. Accessed: 2025-10-11.

The Flutter Team. 2025b. Mocking dependencies with Mockito. https://docs.flutter.dev/cookbook/testing/unit/mocking. Accessed: 2025-11-08.

The Rust Project Developers. 2018a. Associated Types. https://doc.rust-lang.org/rust-by-example/generics/assoc_items/types.html. Accessed: 2025-11-05.

The Rust Project Developers. 2018b. Associated Types. https://doc.rust-lang.org/rust-by-example/trait/supertraits.html. Accessed: 2025-11-05.

The Rust Project Developers. 2018c. Monomorphization. https://rustc-dev-guide.rust-lang.org/backend/monomorph.html. Accessed: 2025-11-05.