IJEEC

International Journal of Electrical Engineering and Computing
Vol. 9, No. 2 (2025)

Original research paper
UDC
DOI

Improving the concept of object-oriented programming in modern programming languages

Jasna Hamzabegović

Faculty of Technical Engineering, University of Bihać, Bihać, Bosnia and Herzegovina

E-mail address: jasna.hamzabegovic@unbi.ba

Abstract— **In software development object-oriented programming (OOP) stands as the principal method because it delivers modular and scalable and reusable code. The traditional OOP implementation models encounter substantial difficulties when managing intricate software projects which resulted in the creation of modern programming languages featuring superior OOP characteristics. This research investigates the modern developments in OOP through assessment of Kotlin together with Rust and Swift along with Scala. The recent improvement packages include trait-based programming along with pattern matching and immutability and Hybrid paradigm unification. Evaluation of software updates takes place at an operational level where efficiency boosts alongside improved robustness and enhanced adaptability receive assessment. Trait-based programming replaces deep inheritance structures, reducing code maintenance complexity. Program safety increases when using Pattern matching combined with type enforcement and the tool simplifies if-then decisions and enables concurrent execution and debugging operations. The research presents an in-depth analysis of new programming language technologies which address classic OOP limitations through higher software development competency improvements.**

Keywords-object-oriented programming, trait-based programming, pattern matching, immutability, functional programming, software engineering

I. INTRODUCTION

The computer programming model known as object-oriented programming maintains its industry significance after its initial introduction during the 1960s. The programming languages Java and C++ together with Python preserve their leadership position in the industry because they let developers encapsulate both object data and behavior. Traditional OOP approaches that combine inheritance along with polymorphism generate two maintenance challenges because they connect code modules and because deep classification details become intricate to handle. The attempt to reach successful modularization has proven unsuccessful.

Overly extensive use of inheritance as a fundamental classical OOP mechanism creates vulnerable base classes that result in harder maintenance of code. Development teams need to manage their deep class hierarchy since parent class modifications can unexpectedly impact the entire child class structure. While polymorphism together with strict encapsulation offers remarkable power, they produce code that becomes more difficult to modify within massive software development projects. Students have difficulty learning OOP due to the abstract concepts of inheritance, polymorphism, and dynamic binding, which are difficult to follow during program execution [1].

The implementation of OOP faces difficulties when applied to distributed and multi-threaded computing systems. A large part of mutable shared state in OOP systems raises parallel programming complexity which produces race conditions and deadlocks [2]. The identification of these issues requires language developers to seek new OOP paradigms and enhancements by adopting functional programming (FP) approaches and alternative methods for modular composition.

Programming languages of modern times have created solutions to tackle these issues though they maintain OOP's fundamental features. Modern programming languages introduce four fundamental features that advance the object-oriented approach — trait-based programming [3], pattern matching [4], immutability [8], and function-based architectures integrating OOP and FP concepts [9]. The current advancements seek to improve OOP by adding features that increase its flexibility as well as scalability and processing capacity in modern software development practice. The paper studies OOP development in contemporary programming languages through comprehensive analysis of solutions that solve previous constraints to enhance software development effectiveness.

II. MATERIAL & METHODS

The research evaluation uses a comparative assessment of OOP improvements found in Kotlin combined with Rust and Swift and Scala programming languages. The research methodology includes three main parts which are: review of available literature, feature comparison and case studies.

A. Literature Review

Multiple academic research papers and industry reports and whitepapers underwent exhaustive review in order to study traditional OOP paradigm shortcomings.

1) Limitations of Traditional OOP Paradigms

Traditional OOP receives wide criticism because its deep inheritance hierarchies together with tight coupling and the fragile base class problem [11]. Research shows that too much code dependency on inheritance results in complex maintainable code because modifications in parents classes trigger unexpected impacts on child classes. The practice of bypassing encapsulation in classical OOP occurs when many programs utilize getter/setter methods which negates the protective advantages of data hiding.

Traditional OOP receives wide criticism because its deep inheritance hierarchies together with tight coupling and the fragile base class problem [11], [12]. Research shows that too much code dependency on inheritance results in complex maintainable code because modifications in parent classes trigger unexpected impacts on child classes [13], [14]. The practice of bypassing encapsulation in classical OOP occurs when many programs utilize getter/setter methods, which negates the protective advantages of data hiding [15], [16].

2) Common Pitfalls in OOP

The research revealed multiple standard problems that emerge when using OOP such as:

- Deep inheritance chains create complexities that result in maintenance hurdles together with code stiffness [10].
- The integration of objects interferes with both modification attempts and testing procedures.
- When managing shared mutable data in concurrent applications state mutability creates race conditions along with deadlock.

3) Advancements in Modern Programming Languages

Programming languages developed in recent times have brought forward innovative methods to handle the existing problems. Table 1 presents the main improvements that have occurred.

TABLE 1: OOP IMPROVEMENTS

Improvement	Description	Example Languages
Trait-Based Programming	Replaces inheritance with modular components	Scala, Rust
Pattern Matching	Enhances polymorphism and reduces complex conditionals	Kotlin, Swift
Immutability	Ensures safer concurrency and reduces side effects	Scala, Kotlin
Hybrid Paradigms	Combines OOP with functional programming for flexibility	Swift, Kotlin

The main purpose of this research investigation focused on building a robust theoretical understanding about fundamental issues in traditional OOP combined with contemporary programming language solutions.

B. Feature Comparison

The research examined OOP concepts in Kotlin Rust Swift and Scala through an organized analysis. The assessment of each language included multiple crucial elements to determine how current paradigms enhance code maintainability and system flexibility and execution speed.

1) Encapsulation and Access Control

Encapsulation functions as the core concept of OOP because it enables data privacy and permission-based access supervision. Java and C++ implement their access control system through private, protected and public modifiers. Members in Swift and Kotlin have both file-private and internal access levels to give developers better granularity in exposure management. Rust runs comprehensive checks on ownership and borrowing to establish both memory safety and continuously protect information from accidental modification.

2) Inheritance and Code Reusability

In Different solutions have been used in the implementation of the latest programming languages which include the following:

- Traits (behavior unit that defines methods) and mixins (a mechanism of combining multiple traits into one class) in Rust and Scala oppose a deep inheritance tree and depict behaviour organization in favour of composition instead of inheritance. In the Scala programming language, the concept of traits enables multiple inheritance of functionality without conflicts, which overcomes some of the limitations of the classic OOP hierarchy [3].
- Protocols with default implementation (for code reuse) essentially gets you protocol-oriented programming for Swift.

3) Polymorphism and Type Safety

Poly-morphism of objects in classical OOP means that whenever type contracts get broken at runtime, we get a runtime error while allowing instances to act as if they are representations of parent classes [17]. Modern languages introduce:

- Kotlin and Scala define pattern matching, which makes instance of checks and explicit casting unnecessary and increases type safety.
- Algebraic Data Types (ADTs) in Scala and Rust are relatively new features that allow for safe type definitions, while minimizing unforeseen runtime issues.

4) Immutability and State Management

The fact that the different OOP systems inevitably make use of mutable state often results in huge problems with debugging and other concurrent programming processes. This is handled in modern languages via:

- The default setting of all data structures is immutable in both Scala and Kotlin unless programmers explicitly waive this limitation.

- Rust enforces data control rules for preventing data race incidents alongside Swift and Kotlin language features which enable valuation definitions with functional properties through their built-in higher-order functions and "val" keyword implementation. Fig. 1 illustrates the conceptual contrast between object-oriented and functional paradigms, showing how modern languages increasingly blend both models to achieve safe concurrency and immutable data flow [10].

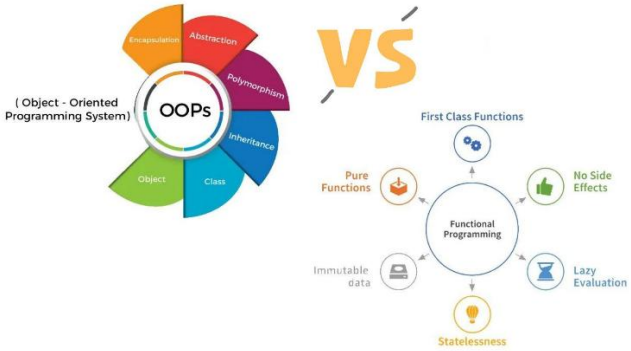

Figure 1. Comparison of Object-Oriented Programming (OOP) and Functional Programming (FP) paradigms [10]

Our evaluation of the research has shown that the modern OOP adaptations provide greater type safety and lead to easier maintenance than traditional development paradigms providing greater flexibility of the system.

C. Case Studies

The research involved multiple case studies to prove how modern OOP programming methods produce functional benefits based on their study findings. Various programming languages and paradigms served as the basis for the case studies that displayed better software creation and operational enhancements. The following approaches were used:

1) Traditional OOP Implementation Using Java and C++

Objective:

A normal software application developed through traditional object-oriented programming in Java and C++ used classic class inheritance with polymorphism features. The developed software contained fundamental banking components to establish new accounts along with money depositions and withdrawals while performing balance calculations [5], [6], [7].

- For the Java implementation process the banking system made use of accounts inheritance to declare different types of bank accounts (e.g. SavingsAccount, CheckingAccount) while also using polymorphism to determine balance calculation methods based on account types [5],[6].
- The C++ version of the banking system utilized inheritance and polymorphism mechanisms that were analogous to the Java version. The program included base classes and derived classes to display standard OOP principles in a statically-typed language structure [7].

Challenges:

- Java together with C++ needed deep inheritance structures that introduced program complexity which reduced design flexibility. Modifying the parent class made developers maintain multiple child classes through inevitable code maintenance problems.
- The inheritance model-maintained duplication between classes despite using polymorphism to reduce it because essential functionality needed a more effective manner of generalization.

2) Modern OOP with Trait-Based Programming and Pattern Matching (Kotlin and Swift)

Objective:

- A restatement of the banking system occurred when developers implemented the same platform again using modern programming languages Kotlin and Swift. Modern programming languages implement traits together with pattern matching which are combined with immutability features in order to overcome OOP traditional restrictions [3].
- Kotlin changed the banking system structure through interface utilization together with higher-order functionality application. The adoption of Kotlin interfaces substituted traits in the code and sealed classes with pattern matching patterns handled various transaction types.
- Swift protocols together with protocol extensions implemented the functions that traditional inheritance would use. The code became more direct through pattern matching together with immutability because these features both cut down the need for conditional statements and mutable state.

Benefits Observed:

- The code reached higher levels of modularity because interfaces (Kotlin) paired with protocols (Swift) allowed behavior mixing through inheritance-free interfaces.
- The type safety of the system improved after Pattern matching became a standard practice in Kotlin and Swift development.
- Utilities of immutable data types within these programming languages generated systems that kept their side effects to a minimum thus creating operational simplicity.

3) Functional-OOP Hybrid Programming Using Scala and Rust

Objective:

The analytic design of a banking system combined FP and OOP methods through its development using Scala and Rust. The designers selected these languages because they combined exemplary features of immutability and functional programming paradigms alongside OOP structures.

- The banking system implemented with Scala made use of sealed traits which defined diverse account types together with transactions. Through its use of immutability and pattern matching capabilities Scala enabled the system to operate transactions of different types with both safety and efficiency.

- Rust utilized Enums to specify account types and its owners' concepts together with pattern matching protocols enabled safe memory operation and prevented processes known as data races.

Benefits Observed:

- The implementation of immutable data types delivered thread safety to the system because threads never shared modifiable states thus minimizing concurrent issues.
- Functional programming concepts which mix with object-oriented principles have enhanced both code flexibility and declaration in the system.

III. RESULTS

A. Trait-Based Programming and Mixins

Current implementations of OOP languages require extensive use of inheritance until problems appear because of deep class hierarchies and code duplication [18]. The design of software becomes complicated and difficult to expand through time when engineers work with this approach. Rust and Scala have implemented trait-based programming as a substitute to resolve inheritance problems by providing reusable components without deep class hierarchies limitations [3] [19].

Outside of interfaces traits enable default method implementations that provide developers with a flexible method of composing behaviors in their code. The implementation removes requirements for numerous inheritance levels to enable reuse of code. The Scala programming language allows dynamic trait integration with classes thus eliminating strict hierarchical limitations.

The code snippet in Listing 1 is written in the Scala programming language, and demonstrates the basic principle of "trait-based programming", i.e. using trait as a code reuse module instead of inheriting from deep classes.

Listing 1. Scala: Trait-Based Programming for Reusable Behavior

```
trait Logger {
  def log(message: String): Unit = println(s"Log: $message")
}
class User(val name: String) extends Logger {
  def greet(): Unit = log(s"Hello, $name")
}
```

The Rust programming language permits developers to establish shared behavior between distinct types with traits after assuring memory safety establishes.

Listing 2. Rust: Defining Shared Behavior Using Traits

```
trait Speak {
    fn say_hello(&self);
}
struct Person;
impl Speak for Person {
    fn say_hello(&self) {
        println!("Hello!");
    }
}
```

Listing 2 illustrates how Rust defines shared behavior through traits. The Speak trait declares a reusable interface, while the Person struct implements it using the impl block (*implementation block*), ensuring type safety and memory safety by design.

Developers who use traits instead of deep inheritance methods obtain better structured code that remains flexible and easy to maintain.

Table 2 clearly shows the shift from a rigid model of inheritance to a more flexible, modular approach in modern languages.

Traits and mixins enable code reuse, simpler hierarchies, greater flexibility, and safer maintenance, thus representing the evolution of OOP towards a combination of functional principles.

TABLE 2: COMPARISON OF TRAIT-BASED PROGRAMMING AND MIXINS

Feature	Traditional Inheritance	Traits (Modern Approach)
Code Reusability	Requires extending base classes	Allows flexible composition of behaviors
Hierarchy Complexity	Leads to deep inheritance chains	Avoids class hierarchy depth
Method Implementation	Inherited methods from base class	Methods can be overridden or mixed in
Flexibility	Limited to single or multiple inheritance	Enables modular and reusable design

B. Pattern Matching and Algebraic Data Types

Pattern matching from both Swift and Kotlin produces polymorphism functionality by cutting down conditional complexity for better code readability. Since traditional OOP relies on if-else statements and switch statements to establish type differences pattern matching provides an organized mechanism that ensures safe processing.

The primary mechanism within Kotlin for pattern matching operates through when expressions that enable developers to conduct strong and secure data type detection. Listing 3 shows a typical example of an algebraic data type (ADT) in languages that blend object-oriented and functional programming.

Listing 3. Kotlin: Pattern Matching over a Sealed Class Hierarchy

```
sealed class Shape
class Circle(val radius: Double) : Shape()
class Rectangle(val width: Double, val height: Double) : Shape()
fun describe(shape: Shape): String = when (shape) {
    is Circle -> "A circle with radius ${shape.radius}" The research methodology includes four main parts which are
    is Rectangle -> "A rectangle with dimensions ${shape.width} x ${shape.height}"
}
```

Scala and Rust manage data in a type-safe way using Algebraic Data Types (ADTs), permitting developers to define a finite set of data structures.

Formally, an algebraic data type defines a new type as a composition of other types or a set of possible constants. A set of alternatives may be enumerated in a type definition, some of which may be recursive. For example, a type can be defined for the four suits of playing cards (spades, hearts, diamonds, clubs), days of the week, the values in Boolean or in Kleene's trivalued logic. The product of these types may also be defined [9]. The program code in Listing 4 demonstrates algebraic data type — a form with two variants (sum type), pattern matching — decomposing values into variants, and type safety and code readability — without an "if/else" chain, all possible variants are covered.

Listing 4. Rust (enum + match): pattern matching over shape variants

```
enum Shape {
    Circle(f64),
    Rectangle(f64, f64),
}
fn describe(shape: Shape) {
    match shape {
        Shape::Circle(r) => println!("Circle with
radius {}", r),
        Shape::Rectangle(w, h) =>
println!("Rectangle of width {} and height {}", w,
h),
    }
}
```

The combination of pattern matching and ADTs through pattern-based expressions, which replace poorly constructed conditional statements, upgrades code expressiveness in a way that modernizes OOP languages [4], as summarized in Table 3. Programmable expressions eliminate if-else statements to have codebases that are not only maintainable but more readable as well. ADTs are statically-typed at compile time, which means strict data integrity is preserved, providing one more advantage to preventing run time errors. Implemented developments benefit developers with higher quality output, and changes in certain debugging features that aid in developing reliable software constructs that detail all structural states and associated behaviour's for all data structures.

TABLE 3: COMBINATION OF PATTERN MATCHING AND ADTS

Feature	Traditional OOP	Pattern Matching & ADTs
Type Handling	Uses instanceof and type casting	Uses structured pattern-matching expressions
Safety	Can result in runtime errors	Ensures compile-time type safety
Readability	Requires long if-else chains	More concise and expressive

C. *Immutability and Functional Integration*

The biggest challenge of Object Oriented Programming is Mutable State, as the interaction of complex debugging scenarios and race conditions leads to unpredictable program behaviour in the context of multi-threaded systems. Simultaneous operation of several states results in malfunction due to their combined action and makes simultaneous performance harder to carry out without failure. Kotlin and Scala were OOP languages that easily provided immutability by preventing changes to assigned data through this mechanism, thereby preventing changes from altering data.

The concept of separating immutable domain logic from the mutable state layer, as illustrated in Fig. 2, aligns with the principles of functional integration discussed in this section. This architecture isolates side effects and enhances thread safety, achieving the immutability goals implemented in languages such as Kotlin and Scala.

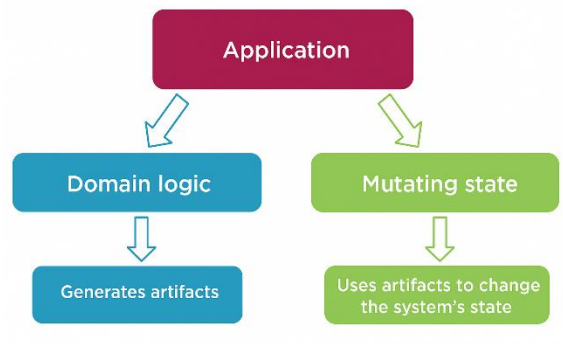

Figure 2. Immutable Architecture [8]

For example, Scala ensures program integrity with "val" declarations that label data constants and disallow mutable collections while Kotlin does this with immutable data classes. The combination of top-level functions with higher-order functions reduce the need for mutable shared state elements thus more predictable and maintainable code. These modern software methodologies add robustness into the development process whilst allowing developers to build scalable, thread-safe applications with reduced reliance on synchronization.

1) *Immutability in Scala and Kotlin*

The val is used because Scala immutability principles dictate that we need to create Immutable variables.

```
val name: String = "Alice" // Immutable
// name = "Bob" // Error: reassignment not allowed
```

As Listing 5 shows, Kotlin data classes are immutable by default.

Listing 5. Kotlin: Immutability in Data Classes

```
data class User(val name: String,val age: Int)
val user = User("John", 30)
// user.age = 31
// Error: Cannot modify immutable data class
```

2) *Functional Programming Integration*

Modern programming languages use higher-order functions along with pure functions to make applications work without changing their underlying state. Rust implements data ownership rules along with borrowing rules to protect against data races, but Swift and Kotlin allow developers to use functional data protection methods. A typical example of functional integration in Swift in Listing 6 shows how pure functions and immutable collections achieve predictable and safe behavior without changing state (immutability).

Listing 6. Swift: example of functional integration

```
let numbers = [1, 2, 3, 4, 5]
let squared = numbers.map {$0 * $0}
print(squared) // [1, 4, 9, 16, 25]
```

These improvements, including better thread safety, code predictability, and simplified debugging, are summarized in Table 4.

TABLE 4. BENEFITS OF IMMUTABLE & FUNCTIONAL OOP

Feature	Mutable OOP	Immutable & Functional OOP
Thread Safety	Requires locks & synchronization	Naturally safe for concurrency
Code Predictability	Mutable state introduces side effects	Pure functions avoid side effects
Debugging Complexity	Harder to track state changes	More predictable code behavior

D. Hybrid Paradigms (Functional-Object Fusion)

The modern programming ecosystem has a variety of hybrid languages that can support use OOP but reap the benefits of FP. Developers gain the ability to produce modular structures that are highly flexible and easy to maintain (while exploiting the benefits of both paradigms) through this combination of paradigms.

Swift and Kotlin are two well-known examples of languages that integrate OOP and FP. It delivers expression quality, conciseness, and improved reasonability in code via first-class functions and higher-order functions (in conjunction with immutable data structures). These languages allow developers to use the functional programming principles to write predictably- and thread-safe structures and preserve the reusability that OOP brings.

Kotlin integrates functional programming through higher-order functions and lambdas in a way that feels very natural to its overall object-oriented structure. With this method, the programmers can write more concise and compact code. The modularity of the blocks of code that higher-order functions generate gives some flexibility that keeps the essential OOP organizational structure intact.

Listing 7. Kotlin: higher-order function implementation

```
class Calculator {
    fun operate(a: Int, b: Int, operation: (Int,
Int) -> Int): Int {
        return operation(a, b)
    }
}

val calc = Calculator()
val sum = calc.operate(5, 3) { x, y -> x + y }
println(sum) // Output: 8
```

Listing 7 shows how functional programming can be implemented with OOP to make things run easier and code simpler.

The language presents developers with the ability to unify OOP methods and functional programming techniques using closures and protocols which produce short and reusable lines of code that retain clear object-oriented abstractions. Swift protocols present an interface pattern that supports default implementations which let developers construct polymorphic patterns that serve both FP and OOP design goals.

1) Main advantages of integrating OOP and FP

The combination of OOP and FP elements during the hybrid development grant developers the privilege to tap into the strengths that each programming paradigm has to offer. The ideal combo of FP immutable data structures and OOP's systematic structure to keep complex systems in check FP's pure functions to minimize side effects of program behaviour plenty of programming advantages come forth from this approach! This hybrid approach allows seamless maintenance as well as scalability, leading to better modular design and programmable composition [see Table 5].

TABLE 5. KEY FEATURES OF HYBRID OOP-FP PARADIGMS

Feature	Traditional OOP	Hybrid OOP-FP
Code Style	Focus on class-based hierarchy	Mixes functions with objects, enabling flexible code blocks
State Handling	Emphasizes mutable state, which can lead to side effects	Encourages immutability and functional purity to avoid side effects
Flexibility	Rigid class structures with tightly coupled components	More modular, composable, and adaptable to changes
Concurrency	Often requires complex synchronization mechanisms	Immutable state and pure functions ensure thread-safety without locks

2) Real-World Application and Benefits

The using object-oriented programming juxtaposition with functional programming enables programmers to write code which is maintainable along their brevity, customization and simplicity to modify as per changing needs. This facilitate functional execution (callback operations and promises), that is passing functions as arguments or in other words using a higher-order functions and closures, while still respecting OOP principles.

It also makes concurrency better. One particularly powerful feature of functional programming is immutable state, which eliminates the salting effect of shared mutable state that provides the majority of race conditions and deadlocks when running code in a multi-threaded manner. Swift and Kotlin have merged paradigms, combining functional and object-oriented features, allowing developers to leverage safety and efficiency while also providing the scalability needed for programming language development.

The Modern programming languages evolve due to the adoption of the OOP constraints solution that helps create more easily maintainable systems, that is more efficient in terms of security and that can scale far beyond the boundaries of complexity and concurrency.

E. Concurrency and State Isolation in Modern OOP-FP Systems

To empirically demonstrate the differences between traditional OOP and modern paradigms, a comparative analysis of implementations in several programming languages was conducted. Table 6 presents a structured overview of common concurrency problems, corresponding language-level mechanisms, and their effects on execution safety and performance. Code listings following the table illustrate how modern paradigms eliminate race conditions and deadlock scenarios without relying on explicit locks..

TABLE 6. OOP CONCURRENCY PROBLEMS AND MODERN SOLUTIONS

Problem in OOP systems	Language solution / mechanism	Effect on competitiveness	Examples of language / techniques
Shared mutable state)	Immutability by default, immutable collections	Eliminates the need for locks, enables deterministic behavior	Scala (val, List), Kotlin (val, List), Swift (let, struct)
Races when accessing facilities (race conditions)	*Actors / Message passing*	Sequentializes access to state without explicit locks	Kotlin Coroutines Channel, Akka (Scala), Swift actor, Rust tokio::mpsc
Deadlock due to multiple locks	*Structured Concurrency* i *async/await*	Avoids circular waiting and deadlock scenarios	Swift Concurrency (async let, TaskGroup),

			Rust async/await, Kotlin coroutineScope
Unpredictable synchronization and complex debugging	*Ownership / Borrowing model*	It statically prevents data races and ensures safe concurrency	Rust (Send, Sync, borrow checker`)
Difficulty maintaining hierarchies and sharing state between instances	*Composition over inheritance*; *Traits / Protocols*	Reduces coupling and facilitates modular testing	Scala traits, Swift protocols, Rust traits
State changes during multithreading (inconsistency)	*Pure functions + state reduction (event → state)*	Deterministic state transformation, without side-effects	Kotlin Flow.scan, Swift Combine.scan, Redux-like models
The need for complex data synchronizations	*Software Transactional Memory (STM)* ili *atomic references*	Guarantees composability and elimination of deadlocks	Clojure STM, Haskell STM, AtomicReference in Kotlinu
Opaque mutation through aliases and references	*Value semantics / Copy-on-write*	Limits access to mutable objects, makes state local	Swift struct (value type), Kotlin data class s copy()

Modern hybrid languages demonstrate that concurrency safety is achieved not by adding synchronization primitives, but by redesigning state management and isolation at the language level.

The following code fragments illustrate selected mechanisms from Table 6 implemented in Kotlin, Rust, and Swift. As shown in Listing 1, messages are processed sequentially, no shared state or locking.

Listing 1. Kotlin: sequential processing without locks

```
class AccountActor(val id: Int, scope:
CoroutineScope) {
    private var balance = 0L
    val inbox = Channel<Msg>(Channel.UNLIMITED)

    init {
        scope.launch {
            for (m in inbox) when (m) {
                is Deposit -> balance += m.a
                is Xfer ->
BankBus.accounts[m.to]?.inbox?.send(Deposit(m.a))
            }
        }
    }
}
```

Listing 2 demonstrates Rust's ownership and borrowing mechanism to statically prevents data races.

Listing 2. Rust (Tokio mpsc): statically prevents data races

```
enum Msg { Deposit(u64), Xfer { to: usize, amt: u64
} }

async fn run(mut bal: u64, mut rx:
mpsc::Receiver<Msg>, peers: Vec<mpsc::Sender<Msg>>)
{
    while let Some(m) = rx.recv().await {
        match m {
            Msg::Deposit(a) => bal += a,
            Msg::Xfer { to, amt } => {
                bal -= amt;
                let _ =
peers[to].send(Msg::Deposit(amt)).await;
            }
        }
    }
}
```

The Swift implementation in Listing 3 ensures state isolation through actors, which guarantee a sequential access — there are no explicit locks or race conditions.

Listing 3. Swift Concurrency: Isolated state with no deadlocks

```
actor Account {
    private var balance = 0
    func deposit(_ a: Int) { balance += a }
    func withdraw(_ a: Int) { balance -= a }
}

actor Bank {
    private var acc: [Int: Account] = [:]
    func transfer(from s: Int, to d: Int, a: Int)
async {
        guard let src = acc[s], let dst = acc[d]
else { return }
        await src.withdraw(a); await dst.deposit(a)
    }
}
```

All three implementations avoid explicit locks. Kotlin and Swift achieve safety via actors and structured concurrency, while Rust enforces race-freedom at compile time through ownership and message passing. As summarized in Table 7, actor-based and value-type paradigms maintain concurrency safety by design. Modern hybrid languages (Kotlin, Swift, Rust) achieve deterministic behavior without explicit synchronization, simplifying debugging and improving scalability.

TABLE 7. COMPARATIVE METRICS OF CONCURRENCY SAFETY

Access	Locks	State mutability	Risk of deadlock	Stability / testability	Note
Java (classic synchronized)	2	High	High	Low	The need for manual locking and access sequencing
Kotlin (actor + Channel)	0	Low (local state)	Low	High	Messages instead of locks
Rust (Tokio mpsc)	0	Low (ownership)	Low	High	Borrow checker eliminates races
Swift (actors)	0	Low	Low	High	Isolated state and sequential await
Kotlin (Flow + ADT)	0	Very low (unchanged)	Low	High	Deterministic flow without side-effects
Swift (Combine + struct)	0	Very low (value-semantics)	Low	High	Value-types instead of shared state

IV. DISCUSSION

The study provides conclusive evidence about modern programming languages which have solved traditional OOP limitations through advanced enhancements. The innovative features of trait-based programming together with pattern matching and immutability provide strong answers to the built-in difficulties within classical OOP through the fusion of

functional programming with object-oriented programming. The recent improvements in OOP through the resolution of deep inheritance chains and mutable states together with polymorphism complexity have transformed the paradigm into a better solution for contemporary software development through enhanced modularity and scalability and improved maintainability.

1) Trait-Based Programming

Code reuse in OOP normally depends on inheritance as its main mechanism yet produces inflexible system structures that restrict future development capabilities. The usage of this method produces highly connected classes and introduces hurdles to software expansion that simultaneously generates unexpected side effects. The programming languages Rust and Scala have combined to present trait-based programming as a solution that replaces the requirements for deep inheritance hierarchies. Traits provide programmers with a mechanism to add functionality to classes through modular design techniques. The implementation of traits as behavior components among developers leads to reduced object dependencies and promotes loose coupling style in software [20].

The Scala programming language enables methods inheritance from multiple traits through traits which prevents users from creating intricate inheritance structures since they can combine diverse behaviors into a single class. The maintenance and flexibility of designs significantly improve when using this method even though traditional inheritance models did not enable the same advanced features.

2) Pattern Matching and Algebraic Data Types

Pattern matching and ADTs are a huge leap because they greatly enhance type safety and expressiveness over OOP. Common OOP polymorphism is based on type casting and instance of checks for implementation but both these methods throw errors which lead to runtime failures. Pattern matching allows developers to define value comparisons using declarative pattern sets within a security programming framework. Kotlin and Swift pattern matching helps programmers as this language feature leads to better polymorphism through an expressive statement pattern that led away from long and bug-prone if-else blocks.

ADTs act as a safeguard for developers to create and manipulate complex data structures. ADTs define all the states a data structure can take and this prevents illegal states thus improving safety and reducing runtime errors. Being able to pattern match exhaustively (Rust Enums combined with Scala sealed classes) means that every case can be caught at compile time, which translates into fewer errors in the code itself.

Fig. 3 demonstrates how an Abstract Data Type (ADT) hides internal data structures (like arrays, linked lists) using public and private functions, exposing only a defined interface to the application program.

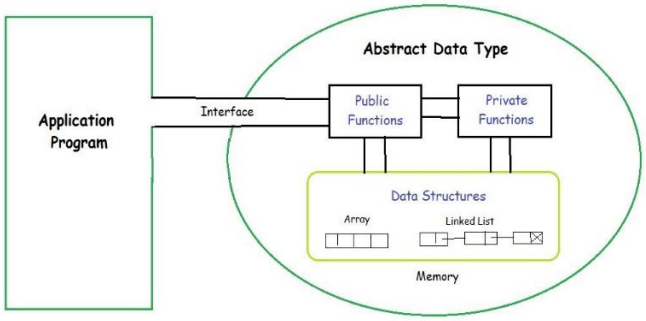

Figure 3. ADT Works [21]

The merge of pattern matching with ADTs enables developers to make their code both readable and maintainable since they do not have to manage difficult type structures while focusing on object behavior control. The additional features enhance debugging capabilities because developers receive instant verification about mismatches or missing cases while performing compilation.

3) Immutability and Functional Programming Integration

In modern OOP languages such as Scala and Kotlin, immutability is promoted as the solution to the issue caused by mutable state. Unexpected effects like races and deadlocks as well as unanticipated responses happen from the storage of mutable data in concurrent systems. Modern programming languages establish immutability as the default gross settings, ensuring that assigned values are modification recall and immutable behaviour thus resulting in consistency and thread safe behaviour.

Since multi-threaded environments do not have any complex resource synchronizing mechanism to keep their data safe from a change, immutability makes data safe by making it non-modifiable. Immutable data enhances program clarity because it enables developers to observe side effects more clearly as well as diminishes the chance of bugs arising from unintended state modifications.

The implementation of pure functions alongside higher-order functions from functional programming yields code which becomes clearer along with being easier to modularize. The programming paradigm with functional characteristics helps programmers create transformations and compositions to manage data flow while properly ensuring that state mutations remain out of their concern. Through Kotlin and Scala developers gain access to a powerful programming combination that lets them use OOP structure together with functional programming flexibility.

4) The Fusion of OOP and FP

Developers now have the advantage of flexible development through the combination of functional programming with OOP because this allows them to select the paradigm which best meets their development requirements. The Swift and Kotlin are two such emerging programming languages that create a powerful synergy between the OOP object abstraction capabilities and the rich expressiveness and composability from FP. This dual-paradigm structure allows developers to switch between the two programming paradigms based on their current needs while creating clean maintainable and extensible code.

This enables designers to cover their complex state requirements through objects using OOP methods, but perform pure computations and concurrency in a manner similar to FP, while composing behaviours with side-effect free expressions. It shows an excellent fit for the transformation of large-scale systems as it offers flexible scalability features, while maintaining robust maintainability characteristics.

Modern programming languages, software developers are able to reap the benefits invisible from both OOP and FP paradigms in a single effort and therefore, create higher-performing and more scalable software frameworks. Through hybrid paradigms developers get the opportunity to write more efficient and cleaner code with less complexity through conventional OOP limitations. The combination of OOP and FP paradigms allows for the development of systems that are simultaneously modular, flexible, and easier to maintain [22].

5) Practical Verification of Concurrency Safety

The empirical results presented in Section III provide practical confirmation of the theoretical assumptions introduced in earlier parts of the paper.

Modern OOP-FP languages such as Kotlin, Rust, and Swift successfully address long-standing concurrency problems inherent to traditional OOP systems — shared mutable state, race conditions, and deadlocks — through built-in language mechanisms rather than external synchronization constructs.

Actors and structured concurrency in Kotlin and Swift, together with Rust's ownership and borrowing model, ensure deterministic message sequencing, state isolation, and compile-time prevention of data races.

This practical verification supports the central thesis of the paper: that hybrid paradigms combining object-oriented and functional concepts not only improve code modularity and expressiveness but also achieve safe, scalable concurrency by design.

V. Conclusion

Empirical verification through Kotlin, Swift, and Rust implementations confirms that modern programming languages achieve concurrency safety without shared mutable state. This represents a major advancement in the evolution of object-oriented programming (OOP), moving toward deterministic, reliable, and maintainable parallel systems.

Modern extensions of OOP mark a new era in software design. Traditional OOP has long faced challenges in scalability and maintainability while attempting to maintain flexibility in large-scale, parallel environments. However, through the integration of trait-based programming, pattern matching, and immutability—key concepts drawn from the functional programming paradigm—modern languages have substantially enhanced OOP's capabilities and resolved many of its longstanding limitations.

Modern applications using trait-based programming, have effectively eliminated inheritance as a traditional base so that developers can create repositionable modules which do not require long inheritance paths, Software developers are enjoying greater flexibility and maintainability as the convoluted hierarchy-inheritance structures have moved to simple reusable traits or protocols. Features like traits of Scala and Rust add balance to the flexibility of a codebase to evolve over time (adding new behaviour with minimal dependencies

Today, most OOP frameworks rely on type casting and instance of checks among others to run well however these mechanisms generate runtime vulnerabilities but significantly hamper their safety levels. In addition to type safety and brevity, pattern matching systems generate human readable code. Through the use of ADTs the system allows only correct data structure states preventing runtime errors and making the developed software more reliable. The new features help improve the reliability of programs by providing stronger ways of expressing both data types and control flow paths in a more secure and less cumbersome way.

The object-oriented programming principles together with functional programming features which Swift and Kotlin have give developers capabilities to create clear and self-explanatory code. Modern languages allow developers to leverage best features from OOP and FP due to their unified implementation of strong object abstraction with structural design and composability and immutability capabilities. Through this combination of paradigms developers achieve complete flexibility because they can select between programming paradigms for each task. Developers should use OOP to address complex state and behavior through objects yet take advantage of FP techniques to create pure code without side effects that enhances testing and maintenance. As Amin et al. [20] point out, understanding the personality traits of software programmers can support the design of educational and development tools tailored to different cognitive styles, which further reinforces the need for more flexible programming paradigms.

The current advancements to object-oriented programming make it much more beneficial for developing modern software. Modern programming languages solve traditional OOP flaws which include deep inheritance structure and mutable objects and tight class coupling so developers can produce better maintainable solutions. Modern OOP becomes more suitable for current software demands while serving as a working base for development of applications that scale efficiently and ensure thread safety and modularity. The development of OOP software depends on the effective fusion between OOP and functional programming to produce safer and cleaner applications. The next research direction should analyze how to improve such enhancements in OOP and establish their integration with current development practices for ensuring OOP's capability to handle modern complex software demands.

The presented comparative implementations and metrics confirm that modern hybrid programming languages achieve concurrency safety through design principles rooted in functional immutability and object encapsulation.

By integrating message passing, ownership semantics, and value-based state management, these languages demonstrate that theoretical models of safe parallelism can be effectively applied in practice. This convergence between theory and implementation marks a significant step in the evolution of object-oriented programming — from imperative synchronization to declarative, deterministic concurrency.

References

[1] J. Yang, Y. Lee, D. Hicks and K. Chang, "Enhancing object-oriented programming education using static and dynamic visualization," *Proc. IEEE Frontiers in Education Conf. (FIE)*, pp. 1–5, 2015, https://doi: 10.1109/FIE.2015.7344152

[2] S. Srinivasan, A. Mycroft and J. Vitek, "Kilim: Isolation-typed actors for Java – A million actors, safe zero-copy communication," in *Proc. ECOOP 2008 – Object-Oriented Programming: 22nd European Conf.*,

vol. 5142, pp. 104–128, Springer, Berlin, Germany, 2008, doi: 10.1007/978-3-540-70592-5_6.

[3] V. P. D. Layka and D. Pollak, "Traits," in *Beginning Scala*, pp. 121–132, Apress, 2015, https://doi: 10.1007/978-1-4842-0232-6_7

[4] S. Ryu, C. Park and G. L. Steele, Jr., "Adding pattern matching to existing object-oriented languages," *Journal of Object Technology*, vol. 9, no. 3, pp. 75–99, 2010, https://doi: 10.5381/jot.2010.9.3.a3

[5] P. Deitel and H. Deitel, *Java How to Program: ATM Case Study Part 2 – Implementing the Design*, Pearson Education, 2017. [Online]. Available: https://www.pearsonenespanol.com/docs/librariesprovider5/2018-college-open-resources/deitel-como-programar-en-java/como-programar-en-java-1e-espcaps-en-linea/capitulo-34.pdf?sfvrsn=525fd2b2_2

[6] "ATM Case Study, Part 1: Object-Oriented Design with the UML," ATM Case Study, Deitel Series, Pearson, 2017. [Online]. Available: https://www.pearsonenespanol.com/docs/librariesprovider5/2018-college-open-resources/deitel-como-programar-en-java/como-programar-en-java-1e-espcaps-en-linea/capitulo-33.pdf?sfvrsn=465fd2b2_2

[7] M. Twain, *Object-Oriented Software Design and Java Programming: Chapter 13 – ATM Case Study Part 2 – Implementing an Object-Oriented Design*, University of Birmingham Press, 2018. [Online]. Available: https://www.studocu.com/en-gb/document/university-of-birmingham/object-oriented-software-design-and-java-programming/chapter-13-atm-case-study-part-2-implementing-an-object-oriented-design/4437434

[8] V. Khorikov, "Immutable architecture," *Enterprise Craftsmanship*, 2016. [Online]. Available: https://enterprisecraftsmanship.com/posts/immutable-architecture/ Accessed: May 7, 2025.

[9] V. Torra, *Scala: From a Functional Programming Perspective—An Introduction to the Programming Language*. Cham, Switzerland: Springer, 2016, https://doi: 10.1007/978-3-319-46481-7

[10] S. Melkonyan, "Object-oriented programming (OOP) vs functional programming (FP)," *Flux Technologies Blog*, 2023. [Online]. Available: https://fluxtech.me/blog/object-oriented-programming-vs-functional-programming/

[11] A. Sabané, Y.-G. Guéhéneuc, V. Arnaoudova and G. Antoniol, "Fragile base-class problem, problem?," *Empirical Software Engineering*, vol. 22, no. 5, pp. 2310–2345, 2017, https://doi: 10.1007/s10664-016-9497-3

Jasna Hamzabegović is an Associate Professor at the Faculty of Technical Sciences, University of Bihać, Bosnia and Herzegovina. She received her B.Sc. degree in Informatics from the University of Sarajevo, the M.Sc. degree in Computer Science and Informatics from the University of East Sarajevo, and the Ph.D. degree in Technical Sciences from the University of Bihać in 2014. Her research interests include educational software development, digital literacy, game-based learning, and user-centered applications for vulnerable populations.

Dr. Hamzabegović has authored or co-authored approximately 40 scientific and professional publications and is the co-author of the university textbook Object-Oriented Programming with C++. She has participated in several international projects in the areas of innovative education and digital transformation and serves as a reviewer for international conferences. She is currently the Head of the Department of Electrical Engineering at the Faculty of Technical Sciences, University of Bihać.

[12] S.H. Tee, "Problems of inheritance at Java inner class," *arXiv preprint*, arXiv:1301.6260, 2013. [Online]. Available: https://arxiv.org/abs/1301.6260

[13] *Naukri Code 360*, "Disadvantages of inheritance in Java," 2023. [Online]. Available: https://www.naukri.com/code360/library/disadvantages-of-inheritance-in-java

[14] *CodiLime*, "Decoding inheritance: An insight into the use and misuse," 2023. [Online]. Available: https://codilime.com/blog/decoding-inheritance-use-and-misuse

[15] M. Skoglund, "A survey of the usage of encapsulation in object-oriented programming," *Department of Computer and Systems Sciences, Stockholm University / Royal Institute of Technology*, Stockholm, Sweden, 2003. [Online]. Available: https://www.researchgate.net/publication/228543013_A_survey_of_the_usage_of_encapsulation_in_object-oriented_programming

[16] Y.Y. Zhuang, W. Kuo and S.C. Tseng, "Resolving the Java representation exposure problem with an AST-based deep copy and flexible alias ownership system," Electronics, vol. 13, no. 2, 350, 2024, https://doi: 10.3390/electronics13020350

[17] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones and A. Spiwack, "Linear Haskell: Practical linearity in a higher-order polymorphic language," *Proc. ACM on Programming Languages (POPL)*, vol. 2, Art. 5, pp. 1–29, 2017, https://doi: 10.1145/315809

[18] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*, Bonn, Germany: MITP-Verlags GmbH & Co. KG, 2015.

[19] *The Rust Programming Language*, "Traits: Defining shared behavior," [Online]. Available: https://doc.rust-lang.org/book/ch10-02-traits.html. Accessed: Apr. 13, 2025.

[20] A. Amin, M. Rehman, R. Akbar, S. Basri and M. F. Hassan, "Trait-based personality profile of software programmers: A study on Pakistan's software industry," in *Proc. 8th Int. Conf. Intelligent Systems, Modelling and Simulation (ISMS)*, pp. 90–94, IEEE, 2018, https://doi: 10.1109/ISMS.2018.00026

[21] *GeeksforGeeks*, "Abstract data types," 2025. [Online]. Available: https://www.geeksforgeeks.org/abstract-data-types/. Accessed: Jun. 1, 2025.

[22] B. M. D. de Sousa, R. C. Ferreira, and A. Goldman, "Functional vs. Object-Oriented: Comparing How Programming Paradigms Affect the Architectural Characteristics of Systems," *arXiv preprint arXiv:2508.00244*, 2025.